On-chip netværk med prioriterede trafikklasser

Thomas Christensen

LYNGBY 2003
EKSAMENSPROJEKT
NR. 37/2003

IMM

Trykt af IMM, DTU

Forord

Denne opgave er resultatet af mit eksamensprojekt udført i foråret 2003. Projektet udgør afslutningen på Civilingeniøruddannelsen på Danmarks Tekniske Universitet (DTU). En tak skal lyde til min vejleder Jens Sparsø for hans entusiastiske og imødekommende vejledning. Yderligere en tak til Shankar Mahadevan og Tobias Bjerregaard, begge Ph.D. studerende ved Institut for Informatik og Matematisk Modellering (IMM).

Lyngby, juni 2003

Thomas Christensen

Resumé

I denne opgave beskrives konstruktionen af et netværk til brug i forbindelse med on-chip systemer.

Netværket opbygges som et todimensionelt gitter, hvor knuderne transmitterer data ved asynkron kommunikation.

Det beskrevne netværk er ikke afhængigt af global synkronisering, og således velegnet til opbygning af GALS (Globalt Asynkront Lokalt Synkront) systemer.

Netværkets knuder udgøres af switche, der gør det er muligt at reservere og prioritere trafik. I fraværet af en fælles global tidsforståelse baseres reservationer i netværket på en tilnærmet form for *time-division-multiplexing*.

Der udføres analyse af latensforholdene for de forskellige trafikklasser. Denne sammenholdes med resultater af simuleringer med forskelligt omfang af baggrundstrafik.

Yderligere gives en vurdering af ressourceforbruget ved implementering af netværket, med hensyn til såvel areal- som energiforbrug.

Abstract

This thesis describes the construction of a network for use in on-chip systems. The network is constructed as a two-dimensional grid, in which the nodes transmits data by asynchronous communication.

The described network is not dependent on global synchronization and therefore well-suited for GALS (Globally Asynchronous Locally Synchronous) systems.

The nodes of the network are switches, by which it is possible to reserve and prioritize traffic. In the absence of a global notion of time reservations in the network is based on an approximated version of time-division-multiplexing.

An analysis is performed of the latency conditions of the different traffic classes. This is compared to results of simulations with varied amounts of background traffic.

Furthermore, an assessment is given, on the use of resources when implementing the network, considering area as well as energy consumption.

Indhold

Figurer

Kapitel 1

Indledning

1.1 Motivation

Den stadige udvikling i chipteknologi med mindre og mindre dimensioner og dermed fortsat voksende kompleksitet i udviklede chips, er sket i et omfang, der ikke er modsvaret af de til rådighed værende værktøjer. Der opstår således et spænd mellem det man evner at fremstille, og det man indenfor en realistisk tidsramme kan designe.

Den store kapacitet i nuværende (og kommende) teknologier har yderligere motiveret en større grad af integration på de udviklede chips, og det er således nu muligt at designe hele systemer på én chip, såkaldte *System-on-Chip* (SoC).

Sammen med stadigt større krav om kort udviklingstid, har den nævnte udvikling betydet en bevægelse i retning af øget genbrug af allerede udviklede funktionsblokke, der kan anvendes til at opbygge det ønskede system.

I sådanne komplekse systemer bliver der behov for kommunikation mellem de enkelte blokke. Primitive løsninger med direkte forbindelser mellem de relevante moduler, eller simple bussystemer, bliver hurtigt upraktiske eller ligefrem uanvendelige. Efterhånden som systemerne bliver større og kommer til at bestå af måske 100 eller flere blokke, bliver det nødvendigt med en mere systematisk opbygning af

kommunikationsinfrastrukturen.

En mulig løsning er at opbygge et netværk på selve chippen, med det formål at forbinde systemets forskellige komponenter. Fordelen ved et netværk er, at der opbygges en regulær struktur, hvis egenskaber bestemmes en gang for alle, og derefter kan genbruges i forskellige designs. Dette er en fordel i forhold til den førnævnte strategi, hvor forbindelsesstrukturen konstrueres specielt til det enkelte design, og hvor en væsentlig arbejdsindsats derfor må lægges i rutning af de nødvendige signaler.

For at kunne drage nytte af de teknologiske muligheder, vi vil få, vil det være afgørende at udvikle en god strategi for opbygning af on-chip netværk, der kan anvendes generelt i forbindelse med *System-on-Chip*. Derfor udgør on-chip netværk et område, der kræver, og i høj grad er berettiget til, en væsentlig interesse. Det er således indenfor dette område, jeg håber at kunne bidrage med dette projekt.

1.2 Formål

Et on-chip netværk ønskes anvendt i mange forskellige sammenhænge, og der kan i det enkelte netværk forekomme trafik af vidt forskellig karakter. Derfor må et godt netværk være i stand til at håndtere denne forskelligartede trafik, på den bedst mulige måde, set udfra en helhedsbetragtning. I den forbindelse er det, som det vil blive beskrevet i kapitel 4, relevant at opdele trafikken i forskellige indbyrdes prioriterede trafikklasser, og f.eks. kunne reservere båndbredde over bestemte forbindelser i netværket. På den måde vil man kunne opnå lavere latens eller større throughput, end det ville være tilfældet i et netværk med kun én trafikklasse.

Målet med denne opgave er at foreslå en løsning til opbygning af et on-chip netværk, der understøtter forskellige trafikklasser, for derved at opnå en service, der kan leve op til de krav om forudsigelighed af latens og throughput, der stilles i et SoC.

Den valgte løsning skal gøre hensigtsmæssig brug af de anvendte ressourcer, i form af anvendt areal og energiforbrug. Samtidig skal løsningen kunne bruges i netværk af forskellige størrelser, uden at det af den grund væsentligt reducerer ydeevnen eller forøger ressourceforbruget uforholdsmæssigt meget.

Ved brug af hardwarebeskrivelsessproget SystemC konstrueres en model af et samlet netværk. Denne model afprøves ved simulering, for at få et billede af den service, der kan tilbydes de tilsluttede enheder.

Dette projekt udgør en indledende undersøgelse, hvorfor ikke alle områder vil kunne blive undersøgt til bunds. I stedet kan det være et udgangspunkt for andre. Det vil således ikke være muligt at give en fuldstændig redegørelse for forholdene i on-chip netværk. Derimod gives der et bud på en løsning, der vil kunne levere de ydelser, der findes mest relevante for et on-chip system.

1.3 Opgavens opbygning

Det antages i det følgende, at læseren har et vist kendskab til netværksrelaterede problemstillinger. Derudover forventes som minimum en forståelse for, hvordan asynkront digitalt design adskiller sig fra traditionelt synkront, og hvilke muligheder og udfordringer dette giver.

Der vil således ikke blive forsøgt at give en fuldstændig udredning af det teoretiske grundlage for on-chip netværk. I stedet fremdrages relevante aspekter, efterhånden som de bliver relevante i forbindelse med overvejelser, der har ført til den valgte løsning.

Efter dette indledende kapitel, vil der i kapitel 2 blive skitseret de omstændigheder, der har gjort det relevant at opbygge on-chip netværk. Her vil der også kort blive motiveret, hvorfor asynkront design er relevant og fordelagtigt til denne type anvendelser. Afslutningsvis beskrives værktøjerne, der har været været anvendt til beskrivelse og simulering.

I projektets tidlige fase blev konstrueret et basalt netværk, hvori kun simpel uprioriteret trafik blev understøttet. Formålet med dette var at få konstrueret de grundlæggende byggeblokke og værktøjer til

senere brug, samt at opnå en bedre forståelse af de mest elementære forhold i denne type netværk. Dette netværk er emnet for kapitel 3, der skaber rammen for vores videre undersøgelser. Her træffes de fundamentale valg for netværkets arkitektur, og der vil i den forbindelse blive redegjort for beslutningerne vedrørende: netværkstopologi, switcharkitektur og rutningsstrategi. På baggrund af disse valg opbygges det grundlæggende netværk, baseret på en simpel switch.

I sidste del af kapitel 3 beskrives konstruktionen af de blokke, netværket er opbygget af, samt af værktøjerne fremstillet til brug ved de indledende afprøvninger og efterfølgende test.

I kapitel 4 argumenteres for indførelsen af forskellige trafikklasser, herunder muligheden for at reservere trafik med henblik på at opnå latens- og/eller båndbreddegarantier. I den forbindelse defineres de trafikklasser, der vil blive arbejdet med i det efterfølgende. Der skitseres desuden, hvilke egenskaber der anses væsentlige for et godt design.

Herefter præsenteres mit forslag til, hvordan et on-chip netværk bør konstrueres, hvis det skal understøtte de beskrevne trafikklasser. Dette indbefatter, udover netværksforhold som databredde og rutningsstrategi, også selve switcharkitekturen. Forslaget baseres på teoretiske betragtninger, samt en vurdering af de acceptable omkostninger mht. ressourceforbrug, afvejet mod den forventede ydeevne.

I kapitel 5 gives et vurdering af løsningen. Først med hensyn til den ydeevne, i form af latens, der forventes for de forskellige trafikklasser. Herefter en diskussion af areal og energiforbrug, der som følge af implementeringens abstraktionsniveau kun kan blive at skøn.

Med netværkets bestanddele beskrevet, vil der i kapitel 6 blive gennemgået de testscenarier, der blev opstillet for afprøvning af netværket. Der vil for hver trafiktype blive undersøgt, hvilken latens data oplever ved forskellige belastninger af netværket. Resultaterne vil blive beskrevet, og der gives efterfølgenden en fortolkning.

I kapitel 7 diskuteres forhold af betydning ved en fysisk implementering. Flere aspekter er allerede behandlet i de foregående kapitler, hvorfor der her er fokus på de forhold, der ikke tidligere er beskrevet.

1.4 Læsevejledning

Der er i denne tekst tilstræbt en opbygning, der i store træk svarer til projektforløbets. Er læseren på forhånd bekendt med SoC-design og Asynkron logik, kan kapitel 2 læses kursorisk.

Selve konstruktionen af netværk og switch gennemgås i kapitlerne 3 og 4, hvor hovedvægten bør lægges på sidstnævnte.

Kapitel 7 er hovedsageligt interessant i forbindelse med videre arbejde mod en fysisk implementering.

Afslutningsvis diskuteres og opsummeres erfaringer, og der gives forslag til videre arbejde indenfor området.

Kapitel 2

Introduktion til fagområdet

Læseren vil i dette kapitel blive introduceret til fagområderne on-chip netværk og asynkront design. Herefter følger en kort beskrivelse af de anvendte værktøjer.

2.1 On-chip netværk

Som allerede nævnt har den øgede kompleksitet af systemer, der implementeres på én chip, ført til en øget segmentering af systemet. Dette er sket både som følge af, at det er blevet muligt at designe større og mere komplekse dele, men også fordi de mere "rummelige"chips har gjort det muligt at samle, hvad der hidtil har været flere forskellige komponenter, på én fælles silicium-die.

Når et system består af adskillige blokke, må der nødvendigvis finde en form for kommunikation sted, for at disse kan arbejde sammen. Det efterhånden meget brugte eksempel er en mobiltelefon, der kunne indeholde blokke som f.eks: et RF-modul til kommunikation med GSM-nettet, andre moduler til lokal kommunikation via infrarød eller Bluetooth protokol, hukommelsesblokke og sidst men ikke mindst naturligvis også et antal mikroprocessorer/signalprocessorer (DSP'er).

Opdelingen i blokke betyder også, at disse kan anvendes i andre

sammenhænge end de oprindeligt er designet til. På den måde opnås en væsentlig besparelse i udviklingstid ved efterfølgende designs, da denne færdige og gennemprøvede blok let kan genbruges i et nyt system. En anden mulighed er at udvikle en generel platform, bestående af et antal standardblokke, der så, konfigureres eller programmeres til den konkrete anvendelse. Her vil der muligvis være dele af den færdige chip, der ikke benyttes. Dette tilsyneladende spild vil dog ofte retfærdiggøres af relativt lavere produktionsomkostninger, der følger af stordriftsfordelene ved at kunne anvende en chip til flere formål.

Med sådanne store systemer vil en væsentlig del af udviklingsarbejdet komme til at bestå i at forbinde de forskellige enheder. Hvis det i det hele taget lader sig gøre, vil denne metode, udover at give en kompliceret forbindelsesstruktur, medføre et ikke uvæsentligt energiforbrug i systemet, til transmission af signaler mellem fjerne enheder.

For i praksis at kunne have sådanne lange signaler, vil det ofte være nødvendigt at indføre buffere undervejs til at forstærke signalet. Disse buffere vil kunne erstattes med *pipe-line* registre for at opretholde en høj clockhastighed. Har man nu etableret lange *pipe-lines* på kryds og tværs af chippen, er skridtet til at forbinde disse i en regulær struktur ikke stort. Hermed har man fået skabt grundlaget for et netværk.

Det er således blevet interessant at konstruere et netværk til brug indenfor en chip. Et sådan on-chip netværk vil forenkle arbejdet forbundet med opbygning af kommunikaitonsinfrastrukturen, og gøre det muligt at genbruge "skabelonen"i forskellige sammenhænge. For at få et godt netværk må netværksarkitekturen være fleksibel og generel nok, til at leve op til de krav der stilles i on-chip systemer. Samtidig dog også så tilpas enkel i sin opbygning, at netværket ikke bliver en dominerende omkostning i det samlede system. Ved omkostning tænkes her primært det chipareal der anvendes til forbindelser og knuder, men også netværkets energiforbrug i drift.

Som følge af de specielle forhold, der gør sig gældende i on-chip sammenhæng, kan man ikke umiddelbart anvende en kendt netværksteknologi. Størst lighed har on-chip netværk med parallelle computere, der allerede er et grundigt behandlet emne i litteraturen. Der er dog stadig væsentlige forskelle.

Den mest markante forskel mellem on-chip netværk og parallelle computere findes i antallet af *wires*, der er til rådighed. Hvor man i multiprocessornetværk er begrænset af antallet af ben på processorkredsen og bus/kabelforbindelsen mellem dem, er det indenfor en chip muligt at have forbindelser af markant større bredde.

I SoC sammenhænge vil man til gengæld ofte være begrænset i energiforbrug. Dette skyldes dels, at der ofte vil være tale om batteridrevne anvendelser, men også interne forhold i chippen ifbm. strømforsyning kan tænkes at spille ind.

Hvad der her betragtes som en god netværksarkitektur er kendetegnet ved at den er:

Fleksibel Netværket skal kunne anvendes i forskellige sammenhænge, og må derfor være i stand til at håndtere den forskelligartede trafik der kan forventes. Fleksibiliteten hos netværket ses både som muligheden for at blive anvendt i forskellige sammenhænge, men også evnen til at håndtere den alsidige trafik der kan forekommer i det konkrete netværk.

Skalérbar Det skal være muligt at opbygge netværk af forskellig størrelse. Arkitekturen må således stadig være anvendelig, selvom behovet for større og mere komplekse systemer øges.

Modulær Udfra fundamentale byggeblokke skal det være enkelt at konstruere et netværk til den givne anvendelse. Yderligere bør netværket hvile i sig selv, hvilket vil sige at det ikke er afhængigt af bestemte tilsluttede enheder. På den måde vil det være enkelt at tilpasse et design blot ved udskiftning af de tilsluttede funktionsblokke.

Det vil således være målet i det følgende, at opbygge et design, der bedst muligt lever op til disse egenskaber.

2.2 Asynkront design

Med de stadigt mindre fysiske dimensioner, og de, som ovenfor beskrevet, heraf følgende relativt større designs, opstår en ny problemstilling. Hastigheden, hvormed kredsløbene på en chip arbejder, bestemmes ikke længere primært af forsinkelsen i logikken, men domineres i stadig større grad af signalernes udbredelseshastighed. Hvor hastigheden af de kombinatoriske kredsløb bliver større i takt med de formindskede dimensioner i nye procesteknologier, vil udbredelseshastigheden af signaler ikke opleve en tilsvarende forøgelse. Desuden sætter lysets hastighed en øvre grænse for signalernes udbredelse.

Transmission af signaler over lange globale forbindelser, vil således ofte tage adskillige clock-perioder. Dette komplicerer selvsagt distribution af en fælles clock til hele systemet. For at undgå at justering af timing-forhold skal blive en alt for omfattende del af et designforløb, vælger man, i stedet for et globalt synkroniseret system, at indføre lokale clock-domæner. Således får man et GALS (Globally Asynchronous Locally Synchronous) system, der forventes at være fremherskende fremover. Denne opdeling falder naturligt i tråd med ønsket om modulær opbygning som diskuteret tidligere, idet de forskellige delsystemer ikke behøver indpasses i et globalt clock-system, men kan have deres eget.

Uden fælles synkronisering må udveksling af data mellem blokke ske ved asynkron kommunikation mellem de forskellige domæner. Dette giver ekstra overhead til den nødvendige *handshaking*, men også øget robusthed, idet der ikke stilles faste krav til timing mellem parterne. Dette gør det enklere at opbygge samlede designs af forskellige delblokke, da enhederne ved passende valg af *handshake*-protokol, blot kan forbindes og kommunikere i det tempo de formår.

Har man indført asynkron kommunikation mellem systemets delblokke har man yderligere muligheden for at gå et skridt videre og implementere et antal af blokkene som egentlige asynkrone kredsløb. Ved asynkront design kan energiforbruget i drift blive større end hvad der kan opnås med et godt synkront design. Den store fordel ved asynkrone kredsløb er, at de ikke kræver en clock, og at energiforbruget i tomgang på den måde kan blive stort set nul. Yderligere er der ikke forsinkelse til opstart af clocken når der igen skal udføres beregninger.

En yderligere fordel ved asynkront design er, at man opnår en ydeevne, der afhænger af de faktiske forhold i chippen, som f.eks. temperatur og forsyningsspænding, i modsætning til synkront design, hvor der designes udfra *worst-case* betragtninger, hvor samtlige beregninger skal nås indenfor den

givne clock-periode, og der derfor må tilføjes en vis sikkerhedsmargin.

Der er her kort nævnt nogle fordele og ulemper ved asynkront design. For en mere fyldestgørende gennemgang af asynkront design henvises til [17].

Vi opsummerer her de væsentligste fordele og ulemper ved asynkront design i forhold til synkront.

Fordele:

- Ingen clock, hvorved der lettere opnås en modulær design strategi.
- Energiforbruget følger arbejdsbelastningen, med heraf følgende lavt tomgangsforbrug.
- Throughput i kredsløb bestemmes af faktiske (aktuelle) forhold i kredsløbet fremfor en *worst-case* vurdering.

Ulemper:

- Større energiforbrug pr. operation/transaktion som følge af den nødvendige *handshaking*.
- Hastighed kan være lavere end optimalt synkront design, pga. *handshake*-signaler, der skal transmitteres frem *og* tilbage.

2.3 SystemC

Netværket, der er konstrueret, er beskrevet i SystemC. Med SystemC er det muligt at arbejde ved et højere abstraktionsniveau end med VHDL eller Verilog. Fordelen ved dette er, at man i designforløbet fra Systemniveau ned til RTL-niveau, gradvist kan raffinere den oprindelige abstrakte beskrivelse, men stadig benytte det oprindelige testmiljø. På den måde undgås en del af de fejl (og besvær), der uvægerligt må følger med konvertering af en model fra et sprog til et andet.

Arbejdet er udført med Synopsys's CoCentric System Studio version 2002.05. System Studio er en grafisk brugerflade, hvormed opnås en visuel repræsentation af det konstruerede design. Dette forbedrer overblikket og letter derfor designforløbet.

Den anvendte version af SystemC er version 2.0. For introduktion til SystemC henvises til [1], [2] og [15].

Efter den indledende installering og konfigurering har jeg, på trods af værktøjets mindre ustabilitet, ikke haft væsentlige problemer ved brugen af hverken System Studio eller SystemC. Da arbejdet hovedsageligt er foregået på relativt abstrakt niveau, og der derfor i praksis stort set har været tale om traditionel C++ programmering, er der dog ikke grundlag for at udtale sig om SystemC's egenskaber som værktøj til egentlig hard- eller firmwareudviklig.

Kapitel 3

Grundlæggende netværk

Der vil i dette kapitel først blive argumenteret for nogle af de mest fundamentale valg der blev truffet i forbindelse med topologi, arkitektur og rutningsstrategi. Herefter vil vi gå over til en kort beskrive af de forskellige delblokke, der er blevet anvendt ved opbygningen af det basale netværk, og som også danner grundlag for det mere avancerede netværk, der er emnet for næste kapitel.

3.1 Topologi

Mange forskellige topologier er gennem tiden blevet foreslået til opbygning af netværk. Den mest umiddelbare måde at forbinde et antal enheder, er ved at trække signaler fra hver enkelt blok, direkte til hver af de øvrige der skal kommunikeres med. Selv for ret begrænsede systemer vil denne løsning hurtigt resultere i et ganske stort antal forbindelser, og vil således blive urealistisk. Et simpelt alternativ er at forbinde de forskellige enheder til en fælles bus. Også denne løsning er problematisk ved større systemer, da kommunikationshastigheden over bussen vil reduceres pga. det store antal porte forbundet til den. Samtidig begrænses trafikken af, at en bus er et delt medie, hvorfor båndbredden der i gennemsnit er til rådighed for den enkelte enhed, er omvendt proportional med det samlede antal.

Ingen af disse muligheder er derfor specielt interessante for netværk af væsentlig størrelse.

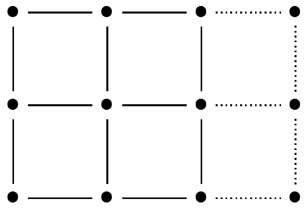
Større kapacitet opnås med eksempelvis en træstruktur, hvor kommunikation i delgren kan foregå uafhængigt af trafikken i de øvrige grene. Der kan dog stadig opstå flaskehalsproblemer, idet al trafikken fra den ene hovedgren til den anden sker gennem træets rod. Dette flaskehalsproblem søges afhjulpet i et Butterfly-netværk, hvor man populært sagt har et træ med mange rødder. På den måder opnås flere alternative forbindelser mellem de enkelte grene.

Mest komplekse er de såkaldte hyper-kuber, hvor netværket opbygges i en flerdimensional struktur. Disse topologier udmærker sig ved at have relativt kort afstand mellem de enkelte knuder. Til gengæld kan de have have et endog særdeles kompliceret sæt af forbindelser. For en mere fyldestgørende beskrivelse af netværkstopologier henvises til [8, kap. 10.4].

Da et egentligt studie af netværkstopologier rækker udover dette projekts område, er det valgt at opbygge netværket som et todimensionelt gitter, som eksemplet i figur 3.1. Knuderne forbindes med fuld duplex *links*, implementeret ved to sæt af unidirektionelle forbindelser, og der gøres dermed udstrakt brug af det store antal *wires* der er til rådighed. Ved at have punkt-til-punkt forbindelser kan kollisioner og deraf følgende tab af pakker undgås, og der spares således energi til eventuel retransmission og areal til implementering af den nødvendige logik.

Valget et gitter som netværkstopologi er truffet udfra følgende betragtninger:

- Et simpelt todimensionalt gitter vil umiddelbart kunne overføres til silicium i ét plan, uden at kræve større indsats i forbindelse med rutning af signaler mellem knuderne.
- Den regulære struktur gør skalering af netværket enkel. Som antydet i figur 3.1 kan knuder tilføjes til kanten af netværket, indtil dette svarer til behovet i den aktuelle anvendelse.



Figur 3.1: Simpelt todimensionelt gitter.

- Bisektionsbåndbredden vokser med tilføjelsen af knuder, hvorfor flaskehalsproblemer er mindre udtalte i et *grid* end i et træ eller en bus.
- Den simple netværkstopologi giver et begrænset antal input og output porte, og dermed enklere knuder.
- Brugen af unidirektionelle *links* gør det muligt at undgå kollisioner og deraf følgende tab af pakker.

3.2 Switcharkitektur

På trods af slægtskabet adskiller on-chip netværk sig på et væsentligt punkt fra multiprocessornetværk. Ved design af On-chip netværk er man ikke i samme omfang begrænses af antallet af *wires* mellem to knuder. Dette skyldes det forhold, at vi kommunikerer indenfor samme chip, og derfor ikke har antallet af ben på chippen til at begrænse os. Det er i stedet fabrikationsprocessen, der bestemmer hvor mange forbindelser vi kan have. Dally og Towles anslår i [10], at der i en 0,1μm CMOS proces vil kunne trækkes 6000 *wires* ud fra kanten af en 3mm bred blok. At dette estimat er foretaget på basis af den minimale wirebredde og -afstand, og derfor ikke levner megen plads til den logik, der skal håndtere signalerne, ændrer ikke på det faktum, at der er mulighed for markant bredere forbindelser end ved

inter-chip netværk.

Det er således antydet, at tilstrækkelig båndbredde ikke umiddelbart er det største problem i et on-chip netværk. Derimod vil buffere, særligt ved disse brede links, kunne udgøre en væsentlig del af arealet ved implementering af en switch. Buffernes formål er at absorbere fluktuationer i netværkstrafikken, således at knuden bedst muligt kan håndtere trafikken. På den måde søges det undgået, at kortvarig ophobning forplanter sig til resten af netværket. Er båndbredden rigelig, vil bufferbehovet være tilsvarende mindre, og arealforbruget til buffering vil så kunne reduceres.

En mulig switcharkitektur kunne bestå af en central hukommelse, hvortil der skrives fra samtlige datainput og læses til de forskellige dataoutput. Denne hukommelsen kan enten implementeres med det nødvendige antal læse- og skriveporte, eller ved på skift at skrive fra de forskellige inputs, og derved "flette"(*interleave*) data.

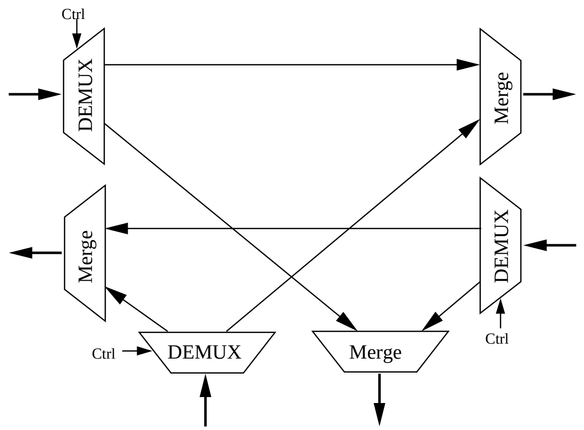
Den første løsning bliver allerede ved et beskedent antal porte både langsom og ressourcekrævende, idet både tilgangstid, areal og energiforbrug er stort ved hukommelser med mange porte.

Alternativet, hvor data flettes sammen, er heller ikke optimalt, idet der her let opstår flaskehalsproblemer, med mindre der er tale om meget hurtig hukommelse.

I et forsøg på, med beskedne ressourcer at opnå størst mulig grad af dekobling mellem de forskellige links til og fra switchen, vælges en DEMUX-Merge arkitektur som skitseret i figur 3.2. Med denne arkitektur opbygges en switch med en DEMUX på hver port. Denne DEMUX har samme antal udgange som switchen, og forbindes evt. via FIFO'er til Merge-enhederne på switchens udgående porte.

Demux-Merge arkitekturen kan være ret ressourcekrævende, da der ved n porte på switchen kræves $n \times (n-1)$ FIFO'er. Merge-enheder med mange indgange er også dyre at implementere, så DEMUX-Merge arkitekturen er kun relevant for relativt små switche.

Vores switch har fem porte, en til den lokale funktionsblok og en for hvert af de fire verdenshjørner. Således vil det kræve 20 FIFO'er, fem DEMUX og Merge enheder med hhv. fire ud- og indgange. Dette



Figur 3.2: DEMUX–Merge arkitektur.

anses for overkommeligt, og vi vælger derfor DEMUX-Merge arkitekturen til vores switch.

Med fastlagt arkitektur og antal porte er størrelsen af FIFO'erne den eneste designparameter, der efterfølgende kan justeres. Længden af FIFO'erne har betydning for, hvor store *bursts* switchen kan håndtere, før trafik i de øvrige retninger påvirkes. Er FIFO'erne for små, vil der ofte ske ophobning med risiko for blokering af de andre links. Til gengæld vil for store FIFO'er optage ekstra plads uden at forbedre netværkets ydeevne.

Da det er målet er at designe en "letvægts-switch, er det undladt at implementere virtuelle kanaler. Virtuelle kanaler kan udnyttes til at forebygge *deadlock* i et netværk, men også til at forbedre throughput ved at mindske risikoen for ophobning, særligt i forbindelse med *wormhole-routing*. For introduktion til virtuelle kanaler henvises til [9].

3.3 Rutningsstrategi

Ved valg af rutningsstrategi er det, særligt ved on-chip netværk, afgørende at vurdere forhold som implementeringskompleksitet og energi-/effektforbrug. Der ønskes således en strategi, der minimerer forbruget af hukommelse og logik i den enkelte knude, og samtidig giver så lav beregningsintensitet som muligt. Herudover er det som altid ønskeligt at opnå et højt *throughput* og den lavest mulige latens.

Rutningsstrategier kan grundlæggende deles op i to hovedgrupper: Dynamiske/adaptive og Statiske. Som ekstempel på en dynamisk rutningsstrategi kendes TCP/IP, hvor der for hvert hop foretages en beslutning om, i hvilken retning data skal sendes videre. Denne beslutning baseres på tabelopslag og eventuelt forespørgsler ud i netværket. Styrken ved en sådan strategi er, at der opnås en stor grad af robusthed overfor ændringer i netværket. Således kan nye forbindelser og nye klienter let tilsluttes, og der kan ved lokale nedbrud findes alternative veje frem til den ønskede destination. Ved en sådan rutningsstrategi, hvor pakkernes vej ikke på forhånd er kendt, kan man ikke være sikker på, at data ankommer i samme rækkefølge som det blev afsendt. Pakker kan have fulgt forskellige veje, eller måske ligefrem være gået tabt undervejs. Det er derfir, i netværk med dynamisk rutning, nødvendigt med mekanismer til at ordne, samle og eventuelt bede om retransmission af tabt data.

Sådanne mekanismer kan være berettigede og rimelige i netværk, hvor der er stor beregningskapacitet, bufferplads og rigelig energi til rådighed. Dette er dog ikke tilfældet i et on-chip netværk, og det vil derfor være nødvendigt at anvende andre strategier, der ikke er så ressourcekrævende

On-chip netværk adskiller sig også ved en større grad af forudsigelighed. Først og fremmest vil topologien, og antallet og arten af de forbundne enheder, være bestemt én gang for alle på designtidspunktet. Der vil således ikke ske løbende tilslutning af nye enheder. Herudover må netværkets bestanddele, f.eks. de forskellige *links*, antages at være til stede og fungere i hele det tidsrum, hvor netværket er i brug(chippens levetid). Dette betyder, at der på forhånd er fuldstændigt kendskab til, hvilke enheder der findes forbundet til netværket, hvor de befinder sig, og muligvis endda også hvilken kommunikation de foretager. Med denne forhåndsviden vil man kunne opnå et fornuftigt resultat med *source-routing*.

Source-routing er kendetegnet ved, at det er afsenderen, der fastlægger datas vej gennem netværket. På den måde bliver funktionen af knuderne i netværket reduceret til at sende data videre i den ønskede retning. De herved sparede tabelopslag vil, udover de førnævnte ressourcer, også spare tid og energi, og dermed mindske både latensen gennem netværker og det samlede energiforbrug.

Ved at anvende *source-routing*, holdes en væsentlig del af kompleksiteten ved rutning hos afsenderne. Denne kan yderligere reduceres, hvis en funktionsblok kun har behov for at kommunikere med ganske få af netværkets enheder. Dens adressetabel kan så begrænses til blot at indeholde de relevante destinationer, fremfor at være en fuldkommen liste over det samlede netværk.
Der er således flere grunde til at vælge en statisk rutningsstrategi:

- Der spares rutningstabeller, og opslag i disse, i hver enkelt knude.
- Afsender har fuld kontrol over transmissionsrækkefølgen, og der spares derved logik og buffering i sender og modtager til at håndtere reorganisering af data.
- Valget af rute træffes af afsenderen, og der gives derfor systemdesigneren stor frihed til at bestemme, hvordan ruter skal vælges. På den måde kan der foretages trafikplanlægning, og forventet tung trafik kan fordeles over forskellige veje gennem netværket.

Ved valg af rutningsstrategi er det af afgørende betydning at sikre, at der ikke forekommer *deadlock*. *Deadlock* opstår som følge af cykliske afhængigheder i netværket, hvor progressionen af data, afhænger

af frigivelsen af ressourcer, der er optaget af data, som (evt. indirekte) venter på progressionen af det oprindelige data. Sådanne cykliske afhængigheder kan, som nævnt brydes med virtuelle kanaler, men kan også helt undgås med en passende rutningsstrategi. Som eksempler herpå kan nævnes: *horizontal-first*, *west-first*, *north-last*, eller variationer af disse. Således kan *deadlock* undgås, hvis der er tale om *store-and-forward* eller beslægtede strategier [8, kap. 10.6].

Store-and-forward rutning er kendt for at kræve meget bufferplads og medføre høj latens, da hele data-pakken skal modtages før den videresendes. For så vidt angår latensen løses dette problem med *virtual cut-through* rutning, hvor flits videresendes så snart den næste knude signalerer, at den vil kunne modtage en hel pakke. I næste knude vil pakken så om nødvendigt kunne samles indtil den kan sendes videre ad næste *link*. Dette står i modsætning til traditionel *cut-through* rutning, hvor en pakke vil kunne være spredt over flere knuder, og dermed kunne blokere for øvrig trafik. I et let belastet netværk vil latensen således kunne reduceres i forhold til *store-and-forward* [8, kap. 10.2.1].

Hvorvidt det er relevant at anvende *virtual cut-through* fremfor *store-and-forward*, vil blive diskuteret yderligere i afsnit 7.2. For denne overordnede undersøgelse er det, for nu, tilstrækkeligt at slå fast, at vi med *store-and-forward* eller tilsvarende, samt hensigtsmæssigt valg af rute, kan undgå *deadlock* i netværket.

Rutningsstrategien i dette netværk er således karakteriseret ved følgende egenskaber:

Source-routing Vejen data skal følge gennem netværket bestemmes af afsenderen

In-order transmission Celler ankommer i samme rækkefølge som de bliver afsendt.

Leveringsgaranti Samtlige celler garanteres at nå frem til destinationen. Der smides således ikke data væk undervejs i netværket.

3.4 Blokke

Der vil i det efterfølgende kort blive beskrevet de funktionsblokke, der er anvendt til konstruktion og test af netværket. Nogle funktionaliteter er ikke umiddelbart relevante for det grundlæggende netværk, men vil blive anvendt i forbindelse med det udvidede netværk i næste kapitel. Udfra disse blokke vil der blive konstrueret et 5×5 netværk, hvori de senere tests vil blive udført.

3.4.1 Simpel pakke

Data transmitteres i simuleringerne som en samlet besked, uden at der af den grund er taget stilling til, hvordan den egentlige transmission skal foregå på det fysiske niveau (se evt. afsnit 7.2.

Den simple pakke, der er beskrevet i filen Simple_packet.h, der kan findes i Bilag C.1, indeholder flg. felter, hvoraf kun få er relevante for det mest grundlæggende netværk, men anvendes ifbm. netværket beskrevet i kapitel 4:

forward_path Angivelse af vejen til destinationen. Opdateres undervejs gennem netværket, således at det altid er de to mindst betydende bit, der angiver retningen til næste hop.

return_path Vejen tilbage til afsenderen. Opdateres tilsvarende undervejs i transmissionsforløbet.

connection_ID Identifikation af den reserverede strøm en pakke eventuelt tilhører.

sender ID der angiver hvem der afsendte pakken.

time_stamp Tidspunktet for cellens afsendelse. Kun relevant ved simulering og test, hvorfor dette felt udelades ved implementering.

data Information der ønskes sendt over netværket. Er her blot et *integer*, men tilpasses pakkens størrelse ved en implementering.

traffic_type Beskrivelse af hvilken trafikklasse cellen tilhører, og dermed også af den information den bærer.

Da pakkerne forventes at have en fast størrelse, vil de,inspireret af sprogbrugen ifbm. ATM, herefter blive benævnt celler. Årsagen til dette vil formentlig blive mere klar i næste kapitel, hvor det grundlæggende netværk tilføjes ekstra funktionalitet.

Koden til Simple_packet findes i Bilag C.1

3.4.2 Abstrakt kanal

Linkene mellem netværkets switche simuleres ved en abstrakt kommunikationskanal. Denne giver mulighed for at udføre datatransaktioner med *send* og *receive* kommandoer, fremfor at forholde sig til en konkret *handshake* protokol.

Udover at give mulighed for en vis abstraktion af kommunikationen over de enkelte *link*, tillader denne model også *mixed-mode* modellering. Dette kan være en fordel under et designforløb, idet der på den måde gives en grænseflade, hvor det er muligt i den ene ende af kanalen arbejdes abstrakt, mens der i den anden ende er valgt en konkret fysisk protokol. Således kan enkelte blokke udskiftes med en udgave beskrevet på et andet abstraktionsniveau.

Med udgangspunkt i koden baseret i [7], er der blevet tilføjet mulighed for at læse data fra den abstrakte kanal uden at afslutte det asynkrone *handshake*-forløbe. Dette er gjort ved at tilføje en *peek*-funktion, og har været nødvendigt for ikke implicit at introducere yderligere buffering i switchene. Med *peek*-funktionen er det muligt at læse og videresende en celle, men først give *acknowledge* til afsenderen, når der er modtaget bekræftelse på modtagelsen. Anvendes *peek*, er det nødvendigt, efterfølgende at afslutte *handshake*-forløbet med et kald til *receive*.

Tilsvarende tilføjelse er gjort til SystemC typen sc_fifo, som så efter denne forfatter er døbt tc_fifo. Tilføjelserne til de to klasser findes i hhv. Bilag C.2 og C.3.

3.4.3 Transmitter

Transmittermodulet emulerer en funktionsblok forbundet til netværket. For at kunne kontrollere trafikken der genereres, kan transmitteren sættes til at arbejde i en af fire tilstande:

1. Celler genereres ved at inkrementere *forward_path* med en fastlagt værdi for hver transmission. Anvendtes ved de helt fundamentale funktionstest af switchen.

2. Tilfældig trafik, forstået således, at der med et givent middelinterval afsendes en celle til en tilfældig modtager. Det nøjagtige afsendelsestidspunkt fastsættes, ved brug af en jævn tilfældig fordeling. Celle n afsendes således indenfor intervallet:

   $$[n * middelinterval; (n+1) * middelinterval[$$

   , hvor n angiver cellens nummer.

   Værdien i *data*-feltet tælles op for hver transmission, og de enkelte celler og deres rækkefølge kan således følges under tesforløbene.

3. Cellernes felter og transmissionstidspunktet fastlægges i udfra beskrivelse i særskilt fil.

4. Kombinerer pkt. 2 og 3, hvorved opnås en datastrøm bestående kendt data blandet med tilfældige celler.

SystemC koden, der beskriver transmitteren findes i Bilag C.4

3.4.4 Modtager

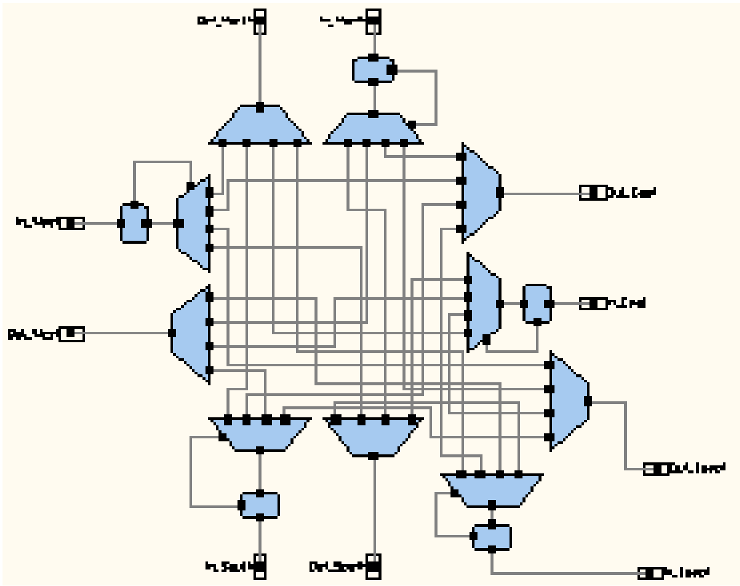
Modtagerenheden giver mulighed for at overvåge hvilke celler der ankommer til den enkelte funktionsblok. I den indledende fase blev denne modtager også brugt ifbm. afprøvning af selve switchen. Det er i modtageren muligt at angive, hvor lang tid der skal gå, inden

næste celle kan absorberes. Denne funktion blev benyttet til at fremprovokere ophobning i netværket.

Output fra modtageren, der kan slås til og fra, gives enten som tekst til *"standard output"*, eller skrives til fil. Sammen med transmitteren gør modtageren det ud for en aktivt kommunikerende funktionsblok tilsluttet netværket.

3.4.5 Simpel switch

Den simple switch er, som tidligere beskrevet, opbygget i en DEMUX–merge arikitektur. Den konkrete opbygning ses i figur 3.3.



Figur 3.3: Opbygning af Simpel switch.

De fem DEMUX'e modtager celler fra de indkomne links. DEMUX'en får sit kontrolsignal fra den lille *switch control*, der desuden opdaterer cellens *forward_path*-felt ved at højreskifte det to pladser.

DEMUX'en sender cellen til den relevante FIFO, hvor den venter til den bliver læst af Merge blokken. Merge blokken, sikrer *fairness* ved cyklisk (*round-robin*) at teste sine inputs for ventende celler. Venter der celler i flere FIFO'er samtidig, vil disse blive sendt hurtigst muligt, og i samme orden som søgerækkefølgen.

Ved spidsbelastning kan der ske en vis ophobning af celler i FIFO'-erne. FIFO'ernes størrelse er da afgørende for hvor robust switchen er for *bursts* i trafikken. Med FIFO'er af endelig størrelse, vil en switch ikke vedvarende kunne håndtere situationen, hvor der sendes mere data til en Merge, end det udgående link kan transmittere. I denne situation vil data ophobes, og en kø vil opstå bagud i netværket. En sådan kø er uheldig, da den kan blokere vejen for andre celler, hvis videre rute har rigelig kapacitet.

FIFO'erne i denne simple switch har her den minimale længde en, da deres primære formål er at undgå, at der opstår kø hver gang to celler ankommer samtidigt (eller inden for en celletransmissionstid) til switchen.

Sålænge den efterfølgende switch modtager data, vil der også kunne sendes fra den aktuelle switch, og der er således altid sikret en vis progression. FIFO'ernes længde er her sat til én. Der vil i det efterfølgende blive beskrevet en mere avanceret løsnining af en switch til on-chip netværk, hvor den simple *round-robin* baserede merge erstattes med et buffermodul med reservationsmekanisme.

Kapitel 4

Netværk med reservation

Dette kapitel omhandler opbygningen af et netværk, hvori det er muligt at håndtere flere forskellige trafikklasser. Det bliver således muligt at foretage reservationer i netværket, og dermed opnå mere forudsigelig latens og throughput.

I lighed med det foregående kapitel vil der her indledningsvis kort blive gennemgået de overvejelser, der har ført til den valgte løsning. Herefter vil selve løsningen og dens funktionaliteter blive beskrevet, efterfulgt af en kort gennemgang af realiseringen i SystemC.

4.1 Baggrund

Hovedfordelen ved et pakkekoblet netværk findes i dets evne til at håndtere trafik af skiftende omfang. En væsentlig ulempe er dog den heraf følgende uforudsigelighed, idet den varierende trafikmængde kan påvirke latensen, der opleves i netværket, væsentligt.

I systemer med realtidskrav vil det ofte være nødvendigt på forhånd at have en god ide om, hvilken latens der kan forventes ved kommunikationen mellem de enkelte enheder.

Lavest latens opnås i et kredsløbskoblet netværk, hvor der før en datatransaktion oprettes en særskilt fysisk forbindelse mellem de kommunikerende parter.

Skal det være muligt at have flere samtidige forbindelser, vil et kredsløbskoblet netværk hurtig blive ganske ressourcekrævende. Et alternativ er *Time-Division Multiplexing* (TDM), hvor forbindelser etableres over delte forbindelser, ved reservation af et antal tidsslots. På den måde kan mange forskellige forbindelser eksistere samtidig over et begrænset antal link.

Ved såvel kredsløbskoblede som TDM netværk er det dog svært at opnå god udnyttelse af den tilgængelige båndbredde. Problemet er, at reserverede tidsslots, selv hvis de ikke udnyttes, ikke er til rådighed for den øvrige netværkstrafik. Derved spildes båndbredde i netværket, der eventuelt kunne have været brugt til transmission af ventende data.

I [14] beskrives en løsning til busarbitrering baseret på TDM, hvor uudnyttede reserverede slots tildeles andre enheder, der ønsker at transmittere data. Således undgås det tab af båndbredde, som er karakteristisk for TDM. I [11] udnyttes ligeledes kombinationen af garanteret service og *Best-effort* til at opnå høj udnyttelse af båndbredden, her på linkene i et egentligt netværk.

Det fremhæves i [11], at muligheden for at kunne reservere trafik vil være af afgørende betydning for et *system-on-chip*, og dermed også en service, der nødvendigvis må tilbydes i et *on-chip* netværk.

En væsentlig fordel ved et netværk, hvor der gives garantier mht. latens og throughput er, at man med den højere grad af forudsigelighed, får reduceret behovet for buffere hos sender og modtager. Ved reservation opnår man desuden, at man før selve transaktionen får viden om belastningen af netværket, og man kan så allerede på det tidspunkt afgøre, om kommunikationen kan gennemføres som ønsket, eller der skal søges andre løsninger.

I et traditionelt pakkekoblet netværk er man overladt til blindt at afsende data, og blot håbe at netværket har den fornødne fri kapacitet. Er dette ikke tilfældet, er der i denne situation ingen mulighed for at undgå overbelastede delstræk, eller på anden måde forebygge situationen. I pakkekoblede netværk er det derfor ofte nødvendigt at indføre mekanismer, der løbende kan om-rute pakker, eller på anden måde dynamisk afhjælpe ophobning i netværket.

Et *on-chip* netværk må forventes at skulle håndtere mange forskellige typer trafik som eksempelvis: Videostrømme, hukommelsesopdateringer, interruptsignaler og *virtual-wires*. Disse datatransmissioner har vidt forskellige omfang, og latenskrav. Det vil derfor være hensigtsmæssigt at definere et begrænset antal trafikklasser, hvori de forskelligartede trafiktyper vil kunne indplaceres.

Uden at gå dybere i diskussionen om, hvilke trafiktyper der kan tænkes at forekomme i et *on-chip* netværk, vil vi i det efterfølgende antage, at *on-chip* netværkstrafik groft kan inddeles i følgende tre klasser:

Strøm Et længerevarende forløb af data, der sendes ad den samme vej mellem en afsender og en modtager, og hvor der foreligger et krav eller ønske om en vis pålidelighed mht. latens og/eller throughput. Den vedvarende karakter betyder, at et vist overhead til et indledende reservations-forløb kan tolereres.

Urgent Små mængder data, hvis hurtige fremførsel gennem netværket er af væsentlig betydning, enten for netværket, afsenderen eller andre delblokke. Eksempler kunne være: Kontrolsignaler, cache-opdateringer, *virtual-wires* eller anden højt prioritet trafik.

Best-effort Øvrig trafik, der ikke stille andre krav, end at data når frem til destinationen i endelig tid, og at indholdet ikke ændres undervejs.

Sammenfattet ønsker vi opbygget en konstruktion, hvor:

- Der kan foretages reservationer med det formål at opnå bedre latens og/eller throughput, end der opnås ved en simpel *best-effort* service.
- Ikke-reserveret trafik kan gives høj prioritet således at vigtige data kan sendes hurtigere gennem netværket end gennemsnittet.

- Netværkskapaciteten udnyttes i videst muligt omfang, der søges undgået retransmissioner eller på anden måde spild af båndbredde.
- Der til stadighed sikres en vis progression i datastrømmen, dvs. *deadlock* og udsultning undgås.
- Energiforbruget holdes så lavt som muligt.
- Arealet af implementeringen holdes så lille som mulig. Dette tilstræbes hovedsageligt ved at minimere brugen af buffere og hukommelse.

4.2 Konstruktion af reservationsswitch

Der vil i det følgende blive redegjort for aspekter af betydning for konstruktion af en reservationsswitch. Reservationsmekanismen, der gør det muligt at prioritere og reservere trafik, beskrives, efterfulgt af en gennemgang af hvordan et reservationsforløb tænkes at skulle foregå.

I lyset af de begrænsede ressourcer der er til rådighed for et on-chip netværk, specielt mht. areal og energi, vil det være væsentligt at søge at begrænse forbruget af disse mest muligt. Hvordan vægtningen, af disse parametre skal være, afhænger af anvendelsen, i hvilken netværket indgår. Vi vil her primært fokusere på lavt energiforbrug, såvel i drift som i hvile, dog med et ønske om et arealforbrug der står i rimeligt forhold til de blokke netværket skal servicere, og dermed systemet som helhed.

Til opbygning af et on-chip netværk vil det således være målet at konstruere en "letvægts-switch", hvor kompleksiteten og energiforbruget holdes så lav som muligt. Dette må nødvendigvis betyde, at der ikke på alle områder kan opnås den ideelle ydelse eller funktionalitet, men forhåbentligt kan der opnås et hensigtsmæssigt kompromis.

Da vi med ønsket om et GALS system har udelukket brugen af en fælles clock, besværliggøres synkronisering i netværket. Uden fælles reference er det ikke muligt at lave *time-division multiplexing*, idet der kan forekomme forskellige opfattelser af, hvilken *time-slot* der er den aktuelle. Synkronisering kunne muligvis opnås ved at sende tomme celler i de perioder, hvor der ikke er anden trafik. Dette er dog ikke hensigtsmæssigt udfra en energimæssig betragtning. Vi er derfor henvist til en tilnærmet løsning, hvor vi i stedet for en fælles global forståelse af *time-slots*, må nøjes med at love, at en given reserveret trafik opnår en bestemt andel af trafikken ud af den enkelte knude. Dette vil være muligt, hvis vi fastholder ideen om at transmittere data som celler, hvor de enkelte pakker har en fast kendt størrelse. Man kan så tælle antallet af celler der afsendes, og på den måde få et tillempet tidsmål.

Er der foretaget reservation til en strøm, og udnytter denne ikke sit tildelte kvantum, må dette ikke være til skade for den øvrige trafik. Derfor må der tilføjes en mekanisme, der sikrer, at den uudnyttede reservation kan bruges til at sendes anden trafik i stedet.

Idet vi har valgt aldrig at dropper celler, og disse desuden garanteres at komme frem i samme orden som de blev afsendt, kan vi bestemme en øvre grænse for latensen i netværket. Grænsen for latensen bestemmes udfra kendskab til switcharkitekturen, hvoraf udledes hvor mange celler, der maksimalt kan være kø af i den enkelte switch. Når yderligere antallet af hop mellem sender og modtager er kendt, er der således fastlagt hvor mange celletransmissionstider en celle maksimalt kan komme til at vente på sin vej til modtageren.

Kø opstår i en switch, hvis der på et givent tidspunkt er celler fra flere inputs, der ønsker at fortsætte videre ad samme link. Er situationen vedvarende, vil ophobning kunne forplante sig i netværket.

Med vores brug af asynkron *handshaking*-protokol håndteres ophobning enkelt og implicit, idet celler vil vente, der hvor de er nået til, indtil det bliver muligt at fortsætte videre.

Det væsentligste problem ved ophobning er, at det ikke vil være muligt at yde cellerne den service de er berettiget til, hvis de står i kø efter celler med lavere prioritet. Prioritering vil først kunne ske, når cellen når frem til en switch, hvor den vil kunne få fortrin over den

øvrige trafik. Dette problem vil kunne afhjælpes med virtuelle kanaler, se evt. [9], hvis disse tænkes anvendt til at adskille de forskellige trafikklasser fra hinanden.

Da virtuelle kanaler vil tilføje yderligere kompleksitet, og muligvis ikke er nødvendige for vores anvendelse, er de ikke anvendt i denne switch.

Reservationsmekanismen

Reservationsmekanismen tænkes op byget omkring en hukommelsesstruktur, bestående af et antal slots, hvor der i hver er plads til en enkelt celle. Disse slots kan reserveres til forbindelser ved en setup-procedure, hvor den enkelte switch afgør om den har mulig hed for at allokere det ønskede antal slots. Udover antallet af slots, vil der ved reservationen også blive angivet, hvor stor frihed switchen har til at udvælge slots til forbindelsen. Et strengt latenskrav må betyde, at de reserverede slots skal være jævnt fordelt, hvorimod der, hvis der blot ønskes et vist throughput, er større frihed til at placere slottene, og om nødvendigt ligefrem samlet. Generelt må det tilstræbes at fordele reservationerne af slots så meget som muligt. På den måde mindskes sandsynligheden for at blokere for senere reservationer der evt. stiller latenskrav.

For at udnytte hukommelsen bedst muligt, anvendes det samme array også til BE-trafik, hvor slots der ikke er reserveret, umiddelbart kan benyttes til øvrig trafik.

For at sikre at BE-trafik altid sikres en vis bufferkapacitet, sættes en øvre grænse for, hvor mange slots der på et givent tidspunkt må være reserveret. Der er ikke på forhånd bestemt hvilke slots der forbeholdes BE-celler, men kun omfanget, for ikke unødigt at reducere muligheden for at gennemføre reservationer før loftet nås.

Ser vi bort fra Urgent-trafik, har vi nu en model, hvor sende- og modtageprocesserne er adskilt af en fælles hukommelse. Data fra inputsiden skrives til slots, afhængigt af om der er tale om celler tilhørende en reserveret strøm, der er tildelt specifikke slots, eller om det er BE-trafik, der placeres i førstkommende frie ikke-reserverede slot.

På outputsiden transmitteres til stadighed cellen i den førstkommende slot, der indeholder ventende data. Er en slot tom søges videre til den næste, med *wrap-around* efter den sidste i følgen. Er samtlige slots tomme, ventes der til næste celle ankommer til switchen og bliver skrevet til en slot.

Når vi yderligere ønsker at håndtere Urgent-trafik, kompliceres sagen en anelse. Det primære formål med at klassificere celler som Urgent er at give dem muligheden for at blive sendt videre, før de BE-celler der allerede venter i switchen. Denne prioritering af Urgent-celler må dog ikke ske på bekostning af den reserverede trafik, hvis planmæssige afsendelse har den absolut højeste prioritet.

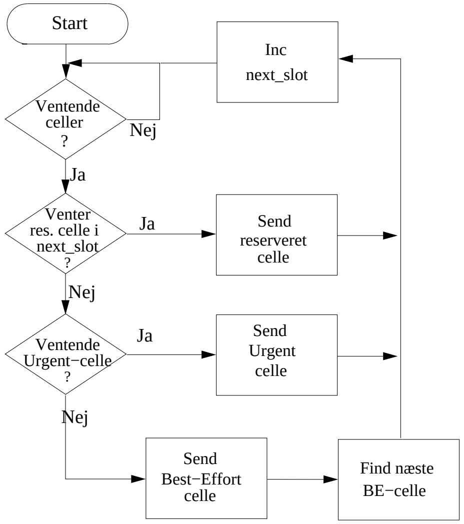
At have en særskilt FIFO til Urgent-celler, og blot sende fra dén når en BE-celle står for tur, er en fristende men ikke acceptabel løsning. Problemet opstår, hvis der i en slot, der kommer efter BE-cellen, venter reserveret trafik. Ved indføjelse af Urgent-cellen, vil både BE-celle og den reserverede celle forsinkes en celletransmissionstid. Dette er ikke acceptabelt, det der skabes unødig usikkerhed om transmissionstidspunktet for den reserverede trafik.

En bedre løsning vil være at holde særskilt styr på, i hvilken slot den næste BE-celle befinder sig. Søgningen gennem slots fungerer grundlæggende som før, men nås der frem til en celle, der enten er tom, eller som indeholder en BE-celle, vil der, i stedet for dén blive transmitteret en celle fra Urgent-køen. Er der ingen ventende Urgent-celler, sendes BE-cellen, der er den næste i tur. På denne måde opnås også, at chancen for at reserverede celler sendes, før det er påkrævet, reduceres, og dermed også risikoen for at generere *bursts*) undervejs i netværket.

Algoritmen til at udvælge cellen, der skal transmitteres, ses skitseret i figur 4.1. Som det ses, sker udvælgelsen dog med den modifikation, at der er en begrænsning på, hvor store følger af Urgent-trafik, der tillades at passere switchen ad gangen. Dette har til formål at undgå lange følger af Urgent-trafik, der vil kunne udsulte BE-trafik,

med den øgede risiko for ophobning dette medfører.

Vi har med denne strategi sikret, at reserverede celler beholder deres retmæssige plads i transmissionsforløbet, og samtidig, at Urgent-celler får mulighed for at blive sendt før BE-celler, dog uden at disse udsultes.



Figur 4.1: Algoritme til udvælgelse af næste transmitterede celle.

Oprettelse af forbindelser

Når en enhed ønsker at oprette en forbindelse, sker det ved at sende et *connection_request* ad den vej, hvor den ønsker at foretage reservationen. I den sendte celle angives det, hvor mange slots der skal reserveres, og hvor stor frihed den enkelte switch gives til at udvælge disse. Som tidligere nævnt vil det, ved krav om lav latens, være nødvendigt kun at give lille grad af frihed, da de reserverede slots ønskes stort set jævnt fordelt. Med større frihed opnås stadig samme throughput, men der kan forekomme større udsving i latensen for den enkelte celle. Udover at angive omfanget af den ønskede reservation, giver afsenderen også forbindelsen et foreløbigt ID, der bruges som foreløbig reference.

På sin vej mod destinationen undersøges *connection_request* cellen i hver enkelt switch. Først afgøres det, om forbindelsen kan oprettes som ønsket. I så fald foretages en foreløbig reservation, med det i cellen angivne ID. Switchen opdaterer cellens *connection_ID*, med en værdi, der angiver i hvilke slots reservationen foretages. Der videresendes således et index, der angiver slots i *afsender*-switchen.

Er der på vejen til modtageren en switch, der ikke kan oprette den ønskede reservation, sender denne et *connection_setup_nack* tilbage til afsenderen. Dette sendes ad vejen beskrevet i cellens *return_path*-felt. Switche, der modtager en *connection_setup_nack*-celle, sletter den foreløbige reservation, den har foretaget, og videresender beskeden.

Et vellykket *connection_request* besvares af modtageren med en *connection_setup_ack*-celle. Denne indeholder et endeligt ID, som modtageren ønsker som identifikation af celler sendt via denne forbindelse.

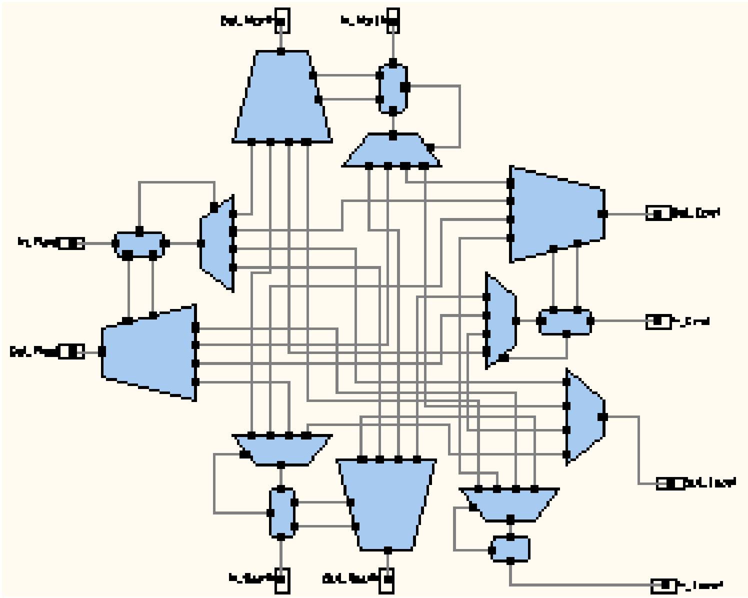
På vejen tilbage til forbindelsens udgangspunkt, bliver de foreløbige ID'er udskiftet med en værdi, der angiver et index for de reserverede slots i den *efterfølgende* switch.

Efter gennemført reservationsforløb kan celler tilhørende den nye forbindelse afsendes, med det ID afsenderenheden fik tildelt, da den modtog sit *connection_setup_ack*. I hvert enkelt hop vil cellernes ID angive, hvilken reserveret strøm de tilhører, og de vil derfor let kunne placeres i den korrekte slot.

Når strømmen er slut, og forbindelsen derfor skal nedlægges for at frigive de reserverede slots, sendes et *connection_teardown*. Dette sletter reservationerne i switchene på vejen mod modtageren. Da celler er garanteret at nå frem, sendes der ikke noget svar til et *connection_teardown*.

4.3 SystemC realisering

Modellen af reservationsswitchen baseres på den simple switch beskrevet i kapitel 3. Denne udvides dog ved at erstatte den hidtidige *Merge* på udgangene med en reservationsbuffer, der indeholder reservationsmekanismen beskrevet tidligere. Da der kan forekomme situationer, hvor celler, der bevæger sig i modsatte retning, kan have betydning for reservationsbufferen, som f.eks. *connection_setup_ack*-celler, og da bufferen også skal have mulighed for at sende data videre ad den vej, er den hidtidige *switching control* enhed tilføjet muligheden for at ekstrahere og injicere celler i datastrømmen. Denne nye blok benævnes *stream control*. Reservationsswitchen som helhed ses i figur 4.2.



Figur 4.2: Opbygning af Reservationesswitch.

4.3.1 Stream_control

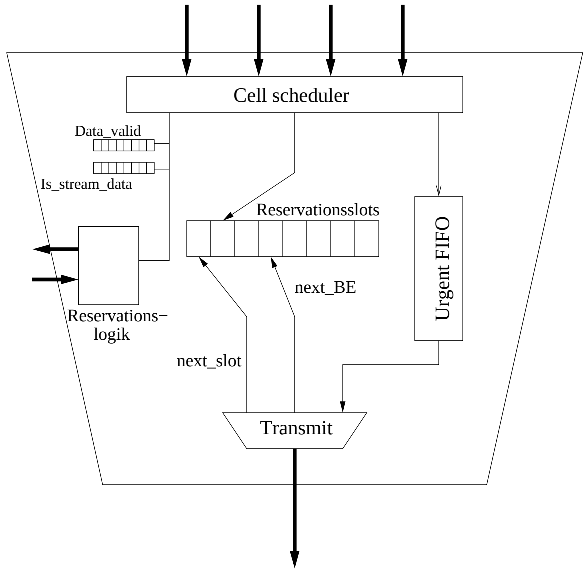
Stream_control modulet overvåger den passerende trafik. Indeholder denne celler af typerne *connection_setup_ack* eller *connection_setup_nack*, bliver disse taget ud og sendt til porten *Packet_extract*, for at blive modtaget af reservationsbufferen. Senere, når cellen er behandlet i reservationsbufferen, modtages den retur via porten *Packet_reinsert*. Da celler, der ankommer på denne port, tidligere er taget ud af strømmen, tillades de at blive flettet ind før en celle der eventuelt ankommer samtidigt på den ordinære indgang. (I asynkron logik vil der dog altid principielt være en orden af handlinger, og dermed ikke samtidighed. Dette skyldes at vi i asynkront design har en kontinuert tidsopfattelse i modsætning til den diskrete, der anvendes ved traditionelt synkront design).

Herudover består funktionen af *Stream_control* i at aflæse den enkelte celles *path* felt, og herudfra generere kontrolsignalet til den efterfølgende DEMUX.

4.3.2 Reservationsbuffer

Reservationsbufferen, der er selve kernen i reservationsswitchen, består som illustreret i figur 4.3 og beskrevet i koden i Bilag C.8, af en *cell_scheduler*, der i *round-robin* orden tager celler fra input-portene og placerer disse i enten en reserveret slot, en fri ikke-reserveret eller i Urgent-FIFO'en, afhængig af hvilken trafikklasse, og eventuelt hvilken strøm cellen tilhører. Denne skedulering baseres på information i de to bit-arrays: *is_stream_data* og *data_valid*, der angiver hvorvidt de enkelte slots er reserveret hhv. indeholder celler, der afventer transmission.

Tildelingen af slots, ved etablering af forbindelse samt afslutningsvis nedlæggelse, håndteres i klassen *reservation_station* (se Bilag C.9). Baseret på de returnerede værdier afgøres det, om *connection_request* skal videresendes med opdateret *connection_ID*, eller om der skal sendes et *connection_setup_nack* i modsat retning. I sidstnævnte tilfælde anvendes cellens *return_path* til at angive vejen



Figur 4.3: Opbygning af Reservationsbufferen.

tilbage til afsenderen. Den oprindelige afsender har, udfra den *return_path* han ser, muligheden for at afgøre, hvor afvisningen kommer fra, og kan bruge denne information til at vælge alternative muligheder eller helt opgive transmissionen.

Ved modtagelse af *connection_setup_nack* (via packet_extract), eller *connection_teardown* nedlægges den relevante forbindelse, hvorefter beskeden sendes videre med passende opdatering af *connection_id*.

På udgangssiden transmitteres celler efter fremgangsmåden illustreret i figur 4.1.

Kapitel 5

Ydeevne og omkostninger

Der vil i dette kapitel blive givet en præsentation af forholdene relateret til ydeevne og ressourceforbrug.

Der vil indledningsvis blive beskrevet latensforholdene i netværket, der kan opdeles i tre kategorier. Disse vil blive beskrevet, og der vil blive udarbejdet estimater for latensen i netværket.

Herefter følger en overordnet diskussion af kompleksiteten af den valgte løsning, og hvilken betydning denne har for forbruget af chipareal og energi.

5.1 Vurdering af latensforhold

Da forholdene i et netværk i drift, herunder samspillet mellem den forskellige trafik, er ganske komplicerede og desuden afhænger meget af den valgte rutningsstrategi, vil der her kun kunne gives et groft overslag over den latensen i netværket.

I det efterfølgende vil der blive søgt givet en dybere indsigt i de parametre og forhold, der påvirker latensen i netværket. Denne indsigt er nødvendig, for at kunne dimensionere netværkets dele til en konkret implementering.

Der vil blive taget udgangspunkt i tre tilstande netværket tænkes at kunne befinde sig i: Massiv ophobning, moderat trafik og svagt

belastet netværk.

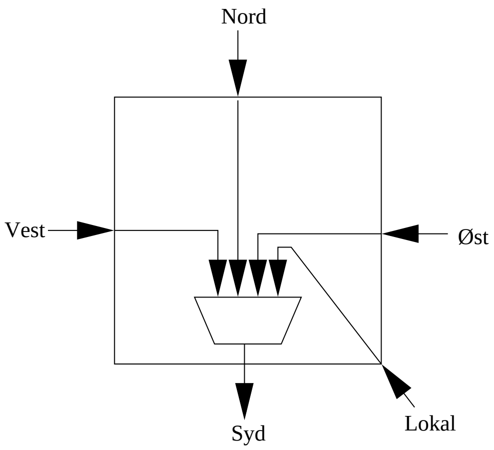
Estimater af transmissionstiden gennem netværket, eller latensen, baseres på den umiddelbare antagelse, at modtagere vil være i stand til at absorbere de celler der bliver sendt til dem. Denne forudsætning må sikres i planlægningsfasen ved at sørge for tilstrækkelig kapacitet og/eller bufferplads i den enkelte enhed, til at denne kan leve op til omgivelsernes forventninger.

Da der, ved passende valg af rutningsstrategi, sikres, at *deadlock* situationer ikke opstår, kan vi argumentere for, at celler i netværket til stadighed oplever en vis progression. Ræsonnementet er induktivt: Når der løbende tages celler ud af netværket, vil der herved skabes plads til at efterfølgende celler kan bevæge sig fremad. Da der ved samtlige sammenfletninger anvendes *round-robin* arbitrering, og der yderligere undgås udsultning, er der således sikret *fairness* for alle indgange til en knude. Da dette gælder generelt, vil alle knuder være garanteret at kunne transmittere celler på enhver af deres porte, i et vist (evt. meget lille) omfang.

5.1.1 Massiv ophobning

Figur 5.1 viser en forsimplet udgave af reservationsswitchen, der illustrerer, hvordan en flaskehalssituation kan opstå i netværket. Tænkes data at ankomme med maksimalt throughput fra vest, nord, øst og den lokale port, og skal al denne trafik samme vej videre mod f.eks. syd, vil der, uanset hvor meget bufferplads der er til rådighed, før eller siden ske en ophobning af data. Ophobningen sker, da der kun kan videresendes en fjerdedel af cellerne, der ankommer til switchen. Sker der ophobning, vil celler, der står i kø udenfor reservationsbufferen, ikke kunne opnå prioritering, før de ankommer til bufferen. Ved en sådan massiv ophobning vil throughputtet på de indgående link reduceres til ca. en fjerdedel af den ideelle kapacitet.

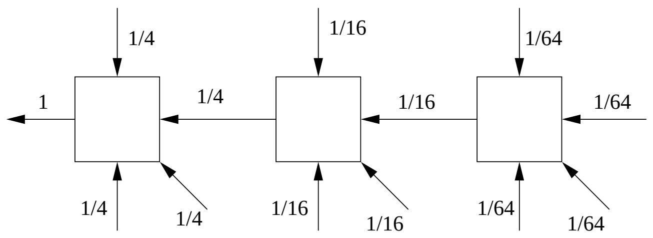
Det er her tydeligt, at der i et *worst-case* scenarie kan opstå alvorlig kødannelse i netværket. Dette er især tilfældet, hvis den beskrevne situation forekommer i flere følgende switche, hvilket hurtigt kan blive konsekvensen af det reducerede link-throughput.



Figur 5.1: Flaskehals i switch

Latensen af celler vil ved massiv ophobning blive meget stor, og som det ses illustreret i figur 5.2 afhænge eksponentielt af antallet af hop en transmission foregår over, idet der ved hvert hop kun opnås en brøkdel af det ønskede throughput.

Denne meget uhensigtsmæssige situation kan og bør undgås ved grundig analyse af netværkstrafikken før design. I den her benyttede gitter-topologi, vil der altid findes et antal alternative veje, data kan



Figur 5.2: Forplantning af ophobning i netværk

rutes ad, og er det ikke muligt at opnå en god fordeling af trafikken, må det overvejes, om netværket er underdimensioneret i forhold til den givne anvendelse, og om linkkapaciteten derfor bør øges.

5.1.2 Moderat trafikintensitet

Har vi et trafikmønster, hvor datamængden, der ønskes transmitteret over de enkelte link, i rimelig grad svarer til kapaciteten, evt. med mindre udsving, der absorberes er switchenes buffere, kan der anslås en øvre grænse for latensen. I denne sammenhæng går vi ud fra, at det er muligt at transmittere celler over det enkelte link med et throughput, der svarer til den maksimale kapacitet.

Ventetiden en celle risikerer at opleve i en switch afhænger af:

1. FIFO-køernes længde.
2. Største afstand mellem tilgængelige reservationsslots (ikke-reserverede slots for BE- eller Urgent-trafik, eller slots reserveret til den givne strøm).
3. Urgent-FIFO'ens længde.
4. Det maksimale antal Urgent-celler der kan sendes i træk (Max_urgent_burst).
5. Transmissionstiden for en celle over det udgående link.

For samtlige typer trafik gælder det, at en celle risikerer at skulle vente på samtlige de celler der er plads til i FIFO'erne, der fører til reservationsbufferen ved den ønskede udgang. Da vi i vores switcharkitektur har fire sådanne FIFO'er, må en celle (let forenklet) vente på, at op til $4 \times FIFO\_size$ celler absorberes af reservationsbufferen.

I reservationsbufferen afhænger ventetiden for de forskellige trafikklasser af forskellige forhold. Ved de efterfølgende estimater af latensgrænser anvendes flg. størrelser, der svarer til værdierne brugt ved simulationerne:

FIFO_size:	1
Urgent_FIFO_size:	3
Antal reservationsslots:	32
Antal reserverbare slots:	30
Max_urgent_burst:	5
transmissionstid:	60
indgangs-/udgangstid:	10
antal hop:	4

Best-effort trafik

Da Best-effort trafik er den lavest prioriterede trafikklasse, vil latensen for denne naturligt kunne blive ganske stor. Antallet af slots, der er tilgængelige for BE-celler, afhænger af, hvor mange reservationer der er foretaget. For at undgå total udelukkelse, er der, fra designerens side, fastsat et maksimalt antal slots, der kan være reserveret. I fravær af Urgent-trafik vil en skeduleret BE-celle, i værste fald først blive sendt, når der er sendt en celle for hver af samtlige reservationsslots. Denne situation opstår, når samtlige andre slots udnyttes, af anden BE-trafik eller reserveret trafik.

Skal der transmitteres Urgent-trafik, vil denne delvist fortrænge BE-trafikken. Som tidligere nævnt er der, for at undgå udsultning, indført en begrænsning på, hvor mange Urgent-celler der kan transmitteres i træk.

En BE-celles latens gennem reservationsbufferen bliver så i værste fald en faktor *max_ burst_ size* større end det før nævnte. Situationen opstår, når der i samtlige slots er skeduleret BE-celler, og der samtidig skal transmitteres en lang følge af Urgent-celler. Sådan en følge bør ikke forekomme (fra samme afsender), men medtages for fuldstændighedens skyld.

Idet vi her går ud fra en situation, hvor der ikke sker ophobning, må den samlede latens fra afsender til modtager forventes at være proportional med antallet af hop. Vi kan således sammenfatte at den øvre grænse for latensen af BE-celler i netværket, er i størrelsesordenen:

$$indg.\ tid + udg.\ tid + ant.\ hop \times trans.\ tid \times$$
$$(ant.\ input \times FIFO\_size + ant.\ slots \times max\_urgent\_burst)$$

Hvilket, ved indsættelse af de relevante værdier, giver:

$$10 + 10 + 4 \times 60 \times (4 \times 1 + 32 \times 5) = 39380$$

Størrelsen af det tilladelige Urgent-*burst* bemærkes her at have væsentlig indflydelse på *worst-case* latensen.

Urgent trafik

Urgent-trafik kan betragtes som højt prioriteret BE-trafik. Dette forstået således, at Urgent-trafik kun kan transmitteres i slots, hvor der ikke er skeduleret reserveret trafik. Urgent-trafik kan dog bryde ind i stedet for ventende BE-celler. Ræsonnementet for øvre latensgrænse er således meget lig det foregående, dog med den udtagelse, at Urgent-celler venter i en særskilt FIFO. En Urgent celle vil i værste fald skulle vente på, at der transmitteres en celle fra det maksimale antal reserverede slots, for hver plads den rykker frem i Urgent-FIFO'en. Vi får således flg. estimat:

$$indg.\ tid + udg.\ tid + ant.\ hop \times trans.\ tid \times$$
$$(ant.\ input \times FIFO\_size + res.\ bare\ slots \times Urgent\_FIFO\_size)$$

Der ved indsættelse giver:

$$10 + 10 + 4 \times 60 \times (4 \times 1 + 30 \times 3) = 22580$$

Reserveret trafik

Latensen for reserveret trafik afhænger hovedsageligt af afstanden mellem de reserverede slots. Største acceptable afstand angives ved oprettelse af forbindelsen, og er derfor kendt. Kan en reservation med de ønskede parametre ikke gennemføres, bliver reservationen afvist.

Da det er afsenderen af den reserverede trafik, der opretter forbindelsen, må vi kunne gå ud fra, at den transmitterede mængde data ikke står i væsentligt misforhold med det reserverede antal slots. Blev en forbindelse misbrugt til at sende mere data end reserveret, kunne dette medføre ophobning, indtil der igen blev en reserveret slot fri.

Øvre grænse for latens ved reservation:

$$indg.\ tid + udg.\ tid + ant.\ hop \times trans.tid \times$$
$$(ant.\ input \times FIFO\_size + res.\ afstand)$$

Giver ved reservationsafstand (mellem slots) på 8 og 2, svarende til reservation af hhv. 4 og 16 slot:

$$10 + 10 + 4 \times 60 \times (4 \times 1 + 8) = 2900$$

$$10 + 10 + 4 \times 60 \times (4 \times 1 + 2) = 1460$$

Det ses således, at der forventes markant bedre afgrænsning af latenserne for reserveret trafik i et netværk uden ophobning.

5.1.3 Lav trafikintensitet

Er trafikintensiteten i netværket (meget) lav, vil celler sjældent være nød til at vente på transmission. I så fald vil cellernes trafikklasse vil være uden betydning for latensen. Dog er det stadig således, at celler tilhørende en reserveret strøm skeduleres i de reserverede slots. På den måde vil senere ankomne BE-celler kunne blive sendt før den reserverede trafik. Dette illustrerer, at der her ikke blot er tale om simpel prioritering af trafikklasserne, men en tilnærmet *Time-Division-Multiplexing*.

Ved lav trafikintensitet vil celler generelt transmitteres med en latens, der ligger nær summen af transmissionstiderne på de enkelte link. Reserveret trafik vil dog have færre helt hurtige celler, og dermed også af denne grund mindre spredning i latensen.

5.2 Ressourceforbrug

Da der er tale om en abstrakt model af netværksswitchen, vil det følgende kun være overordnet diskussion af løsningens kompleksitet, for derved at få et indtryk af areal- og energiforbruget.

5.2.1 Areal

Reservationsswitchen er som beskrevet opbygget af et antal delblokke, der hver især kræver areal i implementeringen.

DEMUX

En DEMUX implementeres billigt i asynkron logik. Ved brug af en *bundled-data* protokol, kræves kun simpel logik til at sende *request*-signalet til den rette modtager. Datasignalerne forbindes direkte til de potentielle modtagere, og kræver således kun rutning af signaler. [17]

Stream control

Stream control indeholder simpel logik, der udfra cellens felter genererer kontrolsignalet til DEMUX'en. Herudover undersøges på tilsvarende vis, om celler skal tages ud af datastrømmen. Dette svarer til en DEMUX funktion som beskrevet ovenfor. Stream control indeholder også en Merge med to inputs til at flette celler tilbage i datastrømmen. I en Merge indgår en MUX, hvis kompleksitet er proportional med cellernes bredde, og koster få logiske elementer pr. bit.

FIFO'er

De i alt 20 FIFO'er i switchen kan, især pga. deres antal, komme til at udgøre en betydende faktor i det samlede arealforbrug. Arealforbruget til FIFO'erne er ligefremt proportionalt med cellestørrelsen, og naturligvis også med FIFO'ernes længde. Arealforbruget til FIFO'er får således flg. sammenhæng:

$$A_{FIFO'er} \propto cellest\o rrelse \times FIFO\_size \times A_{FIFO-bit}$$

Reservationsbuffer

Arealforbruget til reservationsbufferen forventes at være dominerende i reservatonsswitchen. Dette udgøres af flg. delelementer: (se evt. fig 4.3)

Reservationsslots Arealet der må anvendes til reservationsslots hænger selvsagt direkte sammen med antallet af slots, samt af cellernes størrelse, dvs. produktet af cellebredde og antallet af *flits* pr. celle.

Antallet af slots har betydning for granulariteten og antallet af reservationer, der kan håndteres i switchen. Er der kun få reservationsslots vil en enkelt reservation lægge beslag på en stor del af kapaciteten. Yderligere vil muligheden for buffering af BE-celler forringes med den deraf følgende øgede risiko for ophobning i netværket.

Den valgte hukommelsesarkitektur, der benyttes til opbygning af reservationsslottene vil også have betydning for arealet, samt for energiforbruget.

Reservationslogik Kompleksiteten af logikken til at udføre reservationerne, vil i høj grad være afhængig af, hvor raffineret en algoritme der ønskes. En simpel algoritme kunne tænkes anvendt, men en sådan vil formentlig ikke være i stand til at opnå god udnyttelse af de frie slots. Bedre reservation vil kunne opnås med mere forfinede metoder, men må så forventes at være tilsvarende mere bekostelige.

En egentlig undersøgelse af mulige reservationsstrategier er ikke gjort her, men generelt må det forventes, at en iterativ algoritme, der kan implementeres med simpel logik, er at foretrække fremfor kompleks løsning, selvom denne måske er hurtigere.

Dette skyldes, at reserveringen ikke er kritisk mht. tid, da den foregår uafhængigt af den ønskede kommunikation, og et vist *overhead* som før nævnt kan accepteres.

Det er således vurderingen, at reservationslogikken ikke behøver være dominerende i den endelige implementering.

Urgent-FIFO Arealforbruget til Urgent-FIFO'en afhænger på samme måde som de øvrige FIFO'er af cellestørrelsen og FIFO'ernes længde.

Transmitterdelen Består af en Merge, en simpel tæller til *next_slot* samt logik til at bestemme, i hvilken slot næste celle skal hentes. Også her forventes simple iterative løsninger at være at foretrække. Dette skyldes, at udvælgelse af celler til transmission ikke er på den kritiske vej, idet de formentlig vil kunne udføres mens den foregående celle transmitteres, da transmissionstiden forventes at være stor i forhold til beregningstiden.

Bemærk her styrken ved asynkront design, der betyder at beregningstiden bestemmes af den enkelte udregnings "sværhedsgrad", dvs. det nødvendige antal iterationer, og ikke afhænger af *worst-case* situationen.

5.2.2 Energi

Energiforbruget i reservationsswitchen afhænger i overvejende grad af implementeringsforhold. Generelt er der en nær sammenhæng mellem energiforbruget og den datamængde der sendes gennem switchen. Effektforbruget hænger således sammen med det aktuelle throughput.

Der er også enkelte bidrag, der ikke nødvendigvis følger datamængden. Den lineære søgning efter næste celle, der skal transmitteres, bliver f.eks. længere ved sporadisk trafik. Er der slet ingen trafik, må det sikres, at søgningen stoppes, så snart samtlige *data_valid tags* er gennemsøgt. Alternativt kan et signal genereres, der angiver, at samtlige slots er tomme. Søgningen efter celler må dog genoptages, når der næste gang anbringes en celle i en slot.

Er der konstant vekslende trafikintensitet i switchen, vil der, især hvis der er tale om spredt reserveret trafik, forekomme en del søgning. Arbejder switchen derimod ved fuld kapacitet vil søgningerne være korte, og energiforbruget vil være tilsvarende lavt.

Kapitel 6

Tests

Der vil i dette kapitel blive beskrevet de test af netværket, der er blevet gennemført. Formålet med disse har været, at give et billede af latensen de forskellige trafikklasser oplever ved baggrundstrafik af forskellig intensitet.

Indledningsvis defineres det miljø, hvori de efterfølgende tests blev udført. Herefter følger beskrivelser af de enkelte scenarier. I 6.3 gennemgås de opnåede resultater, og de observerede sammenhænge beskrives.

6.1 Miljø

Kvaliteten af den service et netværk yder, er i høj grad afhængig af den trafik, der finder sted i det. Det er derfor ikke muligt at opstille et standardmiljø, der på overbevisende vis beskriver de mange forskellige belastninger, man kan forestille sig. En generel undersøgelse af ydeevne vil derfor i sagens natur være mangelfuld, men er det eneste udgangspunkt vi har.

Idet de mest interessante karakteristika for et netværk vil træde frem, efterhånden som belastningen øges, må der i et relevant testmiljø forekomme en vis grad af "baggrundsstøj". Ved baggrundsstøj forstås trafik, der ikke er relateret til den betragtede kommunikation.

Uden kendskab til enhederne der er forbundet til netværket, eller den rutningsstrategi der måtte være valgt, er vi henvist til at simulere den øvrige trafik i netværket ved tilfældigt genererede transmissioner.

Generering af tilfældig trafik er i sig selv problematisk, da der kræves en "passende"grad af tilfældighed. Den simple model anvendt i transmitterne baserer sig, som tidligere nævnt, på den tilfældighedsgenerator der findes i C++.

Med tilfældighedsfunktionen bestemmes først destinationen for cellen der skal afsendes. Herefter bestemmes det nøjagtige transmissionstidspunkt indenfor en på forhånd angivet tidsramme. Denne tidsramme afhænger af den ønskede trafikintensitet.

Da de tilfældige værdier tilhører en jævn fordeling, får vi en jævn fordeling af kilde - destinationspar. Afsendelsen sker til tider jævnt fordelt indenfor den enkelte celles tidsvindue, og dermed med tidsrammens længde som middelværdi for transmissionsintervallerne.

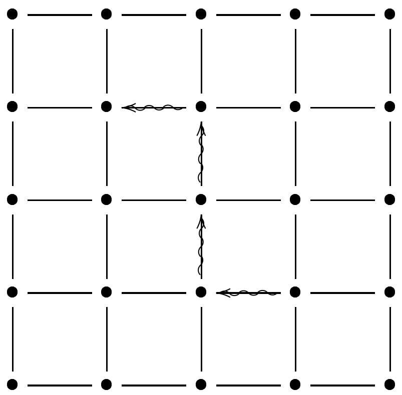
Med et netværk bestående af flere enheder, der på tilfældig vis kommunikerer med hinanden, kunne man forestille sig, at trafikken i netværket også ville fordele sig jævnt. Dette er dog ikke nødvendigvis tilfældet, da det blot er den enkelte kildes destination, der er valgt tilfældigt. Den faktiske fordeling af trafikken i netværket afhænger i høj grad af den valgte rutningsstrategi. Da vi her benytter os af en simpel *horizontal-first* rutningsstrategi, vil der være en væsentlig forskel i belastningen af de enkelte links. (For yderligere diskussion af denne fordeling henvises til Bilag B.1. Her bemærkes det blot, at i et 5×5 netværk opnås ensartet belastning af linkene mellem de ni mest centrale switche, hvorfor vore testscenarier opbygges indenfor dette område.

Vi har således fået fastlagt de omgivelser, hvori de efterfølgende tests vil blive gennemført. Trafikken i dette miljø er fastlagt ved, for hver transmitter, tilfældigt at udvælge destinationen, og tilfældigt bestemme det nøjagtige afsendelsestidspunkt indfor et fastsat tidsrum, der bestemmer trafikintensiteten.

6.2 Scenarier

Generelt for de efterfølgende scenarier gælder, at der udover den observerede trafik forekommer baggrundstrafik af et vist omfang. Sammenhængen mellem transmissionsinterval og belastning udledes i Bilag B.2. Belastningen beskrives som en procentdel af det ideelle throughput over et link, og er således et udtryk for udnyttelsesgraden af dette. Da vi har søgt en jævn fordeling angiver procenttallet baggrundstrafikken over hvert enkelt af de links, hvor testen udføres.

Testcellerne vil yderligere belaste de enkelte links, men søges holdt på et niveau, hvor det ikke bliver dominerende i forhold til netværkets kapacitet, men samtidig af et sådan omfang, at det kan betragtes som en reel datatransmission. Figur 6.1 viser den vej testdata følger gennem netværket.



Figur 6.1: Testdatas vej i netværket.

6.2.1 Best-effort trafik

Som sammenligningsgrundlag for efterfølgende tests, og som billede på et simpelt netværks ydeevne, undersøges den oplevede latens for

Best-Effort (BE) trafik. Ydeevnen bestemmes for at vurdere udbyttet i forhold til omkostningerne ved at tilføje reservationsmekanismen til switchen.

Latensen for 1000 BE-celler bestemmes uden øvrig trafik i netværket. Herefter ved ved 30, 50 og 70% baggrundstrafik, for at belyse den øvrige trafiks indvirkning på latensen. De 1000 celler afsendes med et fast interval svarende til en belastning af de benyttede links på 5% af den maksimale kapacitet. Resultaterne fra simuleringen findes i Bilag D.1 og beskrives i afsnit 6.3.1.

6.2.2 Urgent trafik

Urgent trafik er, som tidligere beskrevet, at betragte som prioriteret Best-effort trafik. Formålet med denne test er at efterprøve, om denne type trafik generelt oplever lavere latens end den ordinære BE-trafik. Ved lav netværksbelastning forventes resultaterne at være sammenlignelige, da der sjældent forekommer ventende data i switchene.

Da Urgent trafik f.eks. kunne være celler indeholdende kontrolinformation, udgøres testforløbet af sekvenser af fire *connection_request* efterfulgt af fire *connection_teardown*. Et *connection_request* må besvares af et *connection_setup_ack*, der afsendes fra destinationen, for at få etableret forbindelsen, der efterfølgende nedlægges. Resultater ses i Bilag D.2 og beskrives i afsnit 6.3.2.

I latensen indgår også tiden, der går med at sætte de relevante reservationsregistre i hver enkelt switch. Da dette, i denne simulering, foregår som et funktionskald uden tidsforbrug, vil det ikke påvirke vore resultater. Den virkelige beregningstid vil selvfølgelig være interessant for en implementering, men er uinteresseant i vores undersøgelse af, hvordan prioriteret trafik i form af Urgent-celler, transmitteres i netværket.

Som i Best-effort scenariet transmitteres 1000 celler uden anden trafik, samt ved 30, 50 og 70% baggrundstrafik. Igen afsendes testceller med en hyppighed, der belaster de relevante links en med på 5% af den maksimale kapacitet.

6.2.3 Reserveret trafik

I et netværk, der ikke oplever ophobning, forventes reserveret trafik at have en latens, der afhænger af, hvor mange slots der er reserveret. I dette scenarie ønskes det at eftervise, hvorvidt tilføjelse af en reservationsmekanisme giver forbedret mulighed for at forudsige latensen af cellerne. Yderligere vil der kunne sammenlignes med estimaterne foretaget i 5.

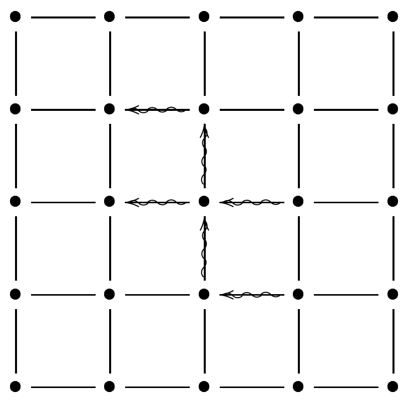
Afprøvningen gennemføres ved indledningsvis at oprette en forbindelse og herefter transmittere 1000 celler med samme interval som i det foregående. Der gennemføres to serier, hvor der reserveres 4 og 16 slots, hvilket i denne simulering, med totalt 32 slots, udgør hhv. 1/8 og halvdelen. Da der reserveres uden frihedsgrad, vil disse slots ligge jævnt fordelt, og vi ved derfor, at ventetiden i reservationsbufferen, for celler der tilhører denne strøm, bør være mindre end 7 hhv. 1 celletransmissionstid. Resultater i Bilag D.3 og afsnit 6.3.3.

6.2.4 Flere reservationer

For at få et billede af interaktionen mellem forskellige reserverede strømme, opstilles to scenarier, hvor reserveret trafik mødes.

Krydsende strømme

To forbindelser der krydser hinanden i en switch, skulle, med den valgte arkitektur, benytte forskellige ressourcer, og derfor ikke påvirke hinandens forløb. Se figur 4.2 på side 36. Scenariet til at efterprøve dette indeholder en hovedstrøm som i foregående scenarie, hvor der reserveres 16 slots. Forbindelse nummer to, der reserverer 4 slots, krydser den første forbindelse, som skitseret i figur 6.2. Ligesom i det foregående scenarie sendes der, efter etablering af forbindelserne, 1000 celler mens der er ingen, 30, 50 og 70% baggrundstrafik. Som resultat forventes latenser, hvis fordeling svarer til dem, der sås uden den krydsende forbindelse.



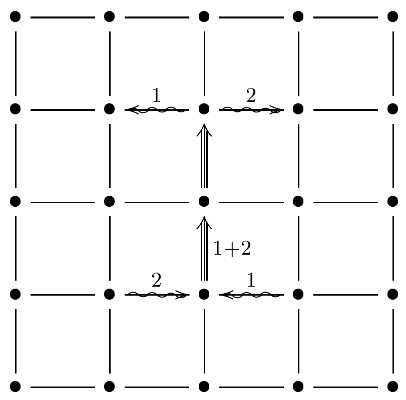
Figur 6.2: Krydsende strømme i netværket.

Overlappende strømme

To strømme, der sendes via fælles links, og dermed deler ressourcer over en del af strækningen, må forventes at ville påvirke hinanden. Formålet her er at eftervise, at:

- Switchen kan håndtere reservation af flere samtidige forbindelser.
- Samtidig reservation sker, uden at strømmenes latenser forværres udover de forventede værdier.

Igen oprettes to forbindelser, som vist i figur 6.3. Forbindelse 1 og 2 reserverer hhv. 16 og 4 slots, og som hidtil simuleres med 1000 celler. Med den øgede trafik over de fælles link viser det sig dog, at netværket overbelastes ved 70% baggrundstrafik. Kørslerne gennemføres derfor i stedet ved ingen, 30, 50 og 65% baggrundstrafik.



Figur 6.3: Overlappende strømme i netværket.

6.3 Resultater

I dette afsnit præsenteres resultaterne af simuleringerne beskrevet i det foregående afsnit. Testresultaterne findes samlet i Bilag D, og der vil løbende blive henvist til disse.

6.3.1 Best-effort trafik

Som forventet ses en konstant transmissionstid i et ubelastet netværk. Dette virker oplagt, da samtlige celler transmitteres straks efter ankomst til hver enkelt switch. Transmissionstiden 275 bliver hermed udtryk for det absolutte *best-case* for en celles latens over fire hop.

Allerede ved 30% baggrundstrafik ses en væsentlig spredning i latensen af cellerne Værdierne ses, for langt størstedelsens vedkommende, at ligge i intervallet mellem 275 og 430, med top omkring 330. En væsentlig del af cellerne (ca. 23%) ses dog at nå gennem netværket uden forsinkelse. Dette stemmer overordentlig godt overens med en forventning om, at der ved 30% belastning over fire hop bør være

$(1-0,30)^4 = 24\%$ sandsynlighed for ikke at skulle vente på øvrig trafik.

Som det kunne forventes vokser latensen gennem netværket, efterhånden som trafikintensiteten øges. Spredningen af tiderne er ligeledes øget, og der ses en langstrakt hale op i store værdier af latensen. Ved 50% baggrundstrafik ses hovedparten af latenstiderne under værdien 700, hvorimod der ved 70% først ses affladning i området over 3000.

Det er således åbenlyst, at latensen af Best-effort traffik påvirkes væsentligt af den omgivende trafik.

6.3.2 Urgent trafik

Resultatet af første testforløb, hvor der simuleres uden baggrundsstøj, kan umiddelbart synes besynderlige, idet der ikke er én latens for Urgent-celler i et tomt netværk. Dette har sin årsag i mekanismen, der skal undgå at BE-trafik udsultes. Da det her er valgt, at der højst kan sendes fem Urgent-celler i træk, hvis der er BE-data der venter. Hvad der ses i figur D.5 er, at hver femte Urgent-celle forsinkes én tidsenhed i hver switch -ialt 4, svarende til den tid det tager i simuleringen at afgøre om der er BE-trafik der venter. Denne situation er speciel for simuleringen i det tomme netværk, da *burst*-tælleren ellers løbende vil blive nulstillet, når baggrundstrafikken transmitteres. Vi har således samme *best-case* latens for Urgent-trafik, som vi havde for Best-effort.

Ved 30% baggrundstrafik ser vi, som ved Best-effort, at en stor del af trafikken når frem uden forsinkelse. Her ses dog en mere jævn fordeling på latenserne, der heller ikke bliver helt så store som ved Best-effort. Forsinkelse af Urgent-celler sker, i fravær af reserveret trafik, kun såfremt en anden celle allerede er i færd med at blive transmitteret.

Ventetiden i en switch, når en sådan opstår, vil således i gennemsnit være en halv transmissionstid. Under normale forhold, hvor der ikke sker ophobning, vil den maksimale latens over fire hop, udover *best-case* værdien, være $4*50 = 200$. Der ses da stort set heller ingen latensværdier over 470 ved 30 og 50% baggrundstrafik, og kun ganske få ved 70%.

Sandsynligheden for, at en Urgent-celle må vente på netop én BE-celle, er ved 30% belastning $4 * 0,30 * (1 - 0,30)^3 \approx 41\%$. Denne masse ses fordelt jævnt over intervallet 275–325. Herefter ses et fald, hvorefter ses de latenser, der er opstået ved interferens med to andre celler.

Er netværksbelastningen stor, må det forventes, at en Urgent-celle skal vente stort set ved hvert hop. Dette giver en middelforsinkelse på $4 \times \frac{1}{2} \times 50 = 100$. Dette ses at stemme fint overens med fordelingen ved 70% baggrundstrafik, hvor vi ser en nydelig klokkekurve omkring latensen 325. Der ses dog enkelte afvigende celler. Ved nærmere undersøgelse af testdata findes disse, at skyldes et enkelt tilfælde af ophobning i netværket.

6.3.3 Reserveret trafik

Ved de indledende forsøg i netværk uden trafik, falder det i øjnene, at transmissionstiderne adskiller sig fra de tilsvarende for Best-effort. Yderligere ses værdierne at afhænge af reservationens omfang. Med fire slots fås en latens på 303, og for 16 slots er den tilsvarende værdi 279. Der er altså tilføjet et overhead på hhv. 28 og 4 tidsenheder, svarende til hhv. 7 og 1 pr. hop.

Forklaringen på den øgede latens skal findes i reservationssystemet, hvor celler sendt som reserveret trafik placeres i den første frie slot, der er reserveret til den. I fraværet af øvrig trafik vil der være et antal tomme slots inden næste slot reserveret til denne strøm. Før en celle kan afsendes må switchen sikre sig, at der ikke venter data tidligere i køen, og det er denne proces, der koster en enkelt tidsenhed pr. slot. Således fås 7 ekstra enheder, når hver 8. slot er reserveret, og blot én når hver anden slot er reserveret til strømmen. Dette fænomen vil være mest markant, ved lav trafikintensitet, da samtlige slots mellem de reserverede celler så vil skulle undersøges. Er den øvrige trafik derimod mere intens, vil index-værdien, der angiver næste pakke der skal sendes, ofte være kommet nærmere den reserverede

slot, inden transmissionen. Denne påstand underbygges af, at der ved de efterfølgende simulationer, hvor der også er baggrundstrafik, ses celler, hvis latens er mindre end i det tomme netværk.

Ikke overraskende har baggrundstrafikkens omfang betydning for middellatensen af de reserverede celler. Fordelingen af latenserne bliver mere symmetrisk, efterhånden som den omgivende trafik intensiveres, samtidig med at maksimalværdierne også vokser. Mest interessant er det dog at se, hvordan reservationens omfang har betydning for det spektret af latenser. Det ses, at latenserne i netværk med samme trafikintensitet, generelt er ca. halvt så store ved reservation af 16 slots, som de er ved reservation af 4, hvilket også svarer til forholdet mellem værdierne anslået i 5.

Det ses, at langt de fleste cellelatenser, hvilket her for ca. 99% af data, holder sig under 1400 og 700 for hhv. 4 og 16 reserverede slots. Sammenlignes dette med Best-effort trafikken, er det helt klart, at der her opnås en væsentlig forbedring i forudsigeligheden af datas transmissionstid. Værdierne ligger også pænt indenfor de de forventede *worst-case* værdier på hhv. 2900 og 1460, med ca. de halve latenser.

Ved reservation af 16 slots ses det, at der er en del celler, der er væsentligt længere om at nå frem end de øvrige. Dette viser sig, som ved Urgent simuleringen, at skyldes ophobning i netværket. At dette fænomen ses her og ikke ved reservation af 4 slots, kan sandsynligvis skyldes, at der er færre ikke-reserverede slots til rådighed for BE-trafikken, dermed bliver risikoen for ophobning større.

6.3.4 Flere reservationer

Krydsende strømme

Af latenshistogrammerne i Bilag D.4 og Bilag D.3, ses at der er god overensstemmelse mellem de to forløb. Ved 70% baggrundstrafik ses der dog, at et noget større antal celler end ved én reserveret strøm, oplever latenser over 1700. Ved nøjere undersøgelse af data ses dette at skyldes nogle få tilfælde af ophobning, og afvigelsen fra det der sås for den reserverede trafik alene, tilskrives derfor tilfældighed, der har gjort at der er opstået flere konflikter i denne kørsel end i den tidligere. Der ses ikke nogen generel tendens, der antyder at krydsende strømme skulle påvirke hinanden.

Overlappende strømme

I et netværk uden anden trafik end de to strømme, hovedstrømmen der har reserveret 16 slots og den overlappende strøm, som kun lægger beslag på fire, ses en forøgelse i latensen som følge af de to strømmes sameksistens. For hovedstrømmen ses en forøgelse på to enheder, hvilket svarer til én for hvert af de to links, hvor der er overlap. Den ekstra latens svarer atter en gang til den ekstra søgetid, der går med at finde frem til den relevante slot.

Den overlappende strøm oplever en latens på 355 mod 303 for en tilsvarende alene i et tomt netværk. Dette er en forøgelse på 52, der må forklares med 2 enheder som skyldes samme fænomen som for hovedstrømmen. De resterende 50 skyldes efter al sandsynlighed, at denne strøm, i vores simulering, hver gang kommer til at vente på cellen fra hovedstrømmen. Dette tilfælde skyldes, at celler i de to strømme afsendes samtidig fra hver deres transmitter, og da der ikke forekommer anden trafik i netværket, vil de også hver gang ankomme samtidigt til switchen, hvor overlappet af strømmene begynder. Fænomenet forventes at have mindre betydning, når der også er anden trafik med i netværket.

Med omgivende trafik ses latensfordelingerne at ligge stort set indenfor samme intervaller som for en strøm alene. Der synes dog at være en tendens til, at fordelingerne er lidt mere jævne, og med tyngdepunktet en anelse længere ude mod de større værdier.

Der er dog også flere celler, der er noget længere om at nå frem end det typiske. Dette må begrundes, dels med den generelt højere belastning af de berørte links, men også med, at der, som også sås ved en 16 slot strøm, er reserveret mange slots i de berørte switche, med deraf følgende færre pladser til BE-trafik.

Generelt ses latenserne stadig at holde sig indenfor et afgrænset interval, selvom vi har flere reserverede strømme over en fælles strækning.

Kapitel 7

Implementeringsaspekter

Der er i det foregående hovedsageligt gjort overordnede designovervejelser. Forhold ved implementering vil dog have afgørende betydning for, hvor velegnet det beskrevne netværk kan blive i praksis. I dette kapitel søges belyst nogle af de overvejelser der er gjort mht. videre implementering.

7.1 Links

Forbindelsen mellem de enkelte knuder er indtil nu blevet betragtet som en abstrakt kanal, hvor de enkelte celler transmitteres med *send* og *receive* kald. På fysisk niveau vil der selvfølgelig være tale om signaler, og der må derfor fastlægges en protokol for dette fysiske niveau.

Bainbridge og Furber gennemgår i [3], forskellige aspekter i relation til design af *delay-insensitive* forbindelser i on-chip systemer, herunder valg af handshake protokol. Her argumenteres for anvendelse af en 1-af-N dataenkodning med RTZ (*Return-To-Zero*) signalering, da dette giver få signaltransitioner ifht. transmitteret data, og samtidig simplere kredsløb end *transition-signalling*.

For ikke at få for mange inputs, og dermed sløve logikken til generering af *acknowledge*-signaler, kan det være hensigtsmæssigt at

opdele linket i et antal *gangs*. Disse *gangs* gennemløber hver deres individuelle signaleringsforløb.

Undlader man at synkronisere signalerne for hvert *pipe-line* trin, men blot for enden af linket, vil der opnås en lavere middellatens. Dette sker, da den statistiske fordeling af latenserne ændres, da den ikke længere er udgøres af summen af et antal ekstremværdiprocesser, hvor det langsomste *gang* bestemmer hastigheden i hvert trin. I stedet er der nu tale om summering af (uafhængige?) stokastiske variable, der kun har ekstremværdifordeling ved den afsluttende synkronisering. På den måde kan et *gang*, der er "kommet dårligt fra start"få chancen for at "vinde ind"på de øvrige i de efterfølgende *pipe-line* trin, fremfor at sinke alle.

Førnævnte forfattere giver i [4] et eksempel på en anvendelse af nævnte signalering i deres Chain netværk. Der gives her eksempler på, hvordan pipe-line latche, arbitere, og en simpel ruter kan konstrueres. Det anslås desuden, at der i 0,35 og 0,18mym CMOS teknologi, kan opnås throughput på hhv. 120 og 160 Mb/s pr. leder.

I valget af linkdesign er det væsentligt at bemærke, at dataenkodning og signaleringsstrategi giver mulighed for at opnå besparelse i energiforbrug mod at anvende flere *wires*, en ressource der som tidligere nævnt er billig i et on-chip miljø. Således gives her parametre, der muliggør optimering i forhold til den konkrete anvendelse.

7.2 Celleformat

Celleformatet bestemmes i det væsentligste ved at fastlægge antallet af flits, og bredde af disse. Bredden af flits begrænses af de fysiske dimensioner, men den optimale størrelse kan kun bestemmes ved samtidig at vurdere omkostningen af den nødvendige hardware i netværkets forskellige dele.

Brede flits giver større throughput, og hvis cellen i stedet for at bestå af flere flits, udgøres af en enkelt flit med tilsvarende større bredde, vil dette betyde mindre latens og enklere logik i switchene.

Består en celle af flere flits, må der i switchen løbende holdes styr på antallet af disse, eller der må, som i [4], indføre end-of-packet (eop) signaler. Sidstnævnte for at få simplere kredsløb på bekostning af mere *overhead* i datatransmissionen.

I bestræbelserne på at konstruere en "letvægts-switch vil det være optimalt at have celler, der kun består af en enkelt flit. På den måde vil vi få et *store-and-forward* netværk, og ikke behøve mekanismer til at implementere *virtual cut-through* rutning. Vælger vi en sådan enkelt-flit celle, må den dog have en vis bredde, for at opnå et fornuftigt forhold mellem datafeltets og cellens størrelse, dvs. et acceptabelt *overhead*.

Felterne i celler fra forskellige trafikklasser behøver ikke være . For at kunne have effektiv logik i switchene, er det dog vigtigt, at felter, der skal læses eller modificeres undervejs, har samme placering i de forskellige typer celler. Felter der ikke udnyttes, som f.eks. *Stream_ID* i en BE-celle, kan dog inddrages som datafelt.

Der kræves således en vis indsigt i de fysiske forhold omkring implementering og viden om de konkrete krav til netværket, for at kunne bestemme det mest hensigtsmæssige celleformat. Der ligger såleds her en væsentlig udfordring ved design af det endelige kredsløb.

7.3 Interfaces

Der er ikke her gjort videre overvejelser mht. interfacet mellem netværket og de tilsluttede funktionsblokke. Det vil ved videreudvikling være naturligt at konstruere et generelt interface, der kan bruges som grænseflade til disse blokke. Interfacet kan, men behøver ikke, følge en vedtagen standard, som f.eks. OCP. I mange situationer vil det være hensigtsmæssigt med et simpelt interface, der blot implementerer de funktioner der er relevante for den givne blok. Således bruges der ikke arealet til uudnyttede funktionaliteter. Ovenpå den simple grænseflade vil der så, om nødvendigt, kunne tilføjes yderligere lag , der gør det muligt at forbinde standardblokke udviklet andetsteds.

Kapitel 8

Diskussion

Der er i de foregående kapitler blevet foreslået en løsning til en netværksarkitektur, hvor det er muligt at prioritere trafik. Latensforholdene i dette netværk er indledningsvis blevet analyseret, og der er efterfølgende gennemført et antal simulationer for at afprøve funktionaliteter og for at få et billede af den opnåede ydeevne.

Resultaterne viser, at den oplevede latens i netværket, med ganske god margin, ligger indenfor det forventede. Det må dog i den forbindelse bemærkes, at der ved de gennemførte tests ikke er forsøgt at fremprovokere de *worst-case* scenarierne, der er grundlaget for latensoverslagene. De fremførte resultater må således ikke tages som bevis for faktiske maksimalværdier, men derimod mere som en illustration af, at trafikken i netværket prioriteres som ønsket, og at det på den måde er muligt at differentiere sin netværksservice, efter kommunikationens art.

Det er med den udviklede model muligt at sende Urgent-data, der med høj prioritet transmitteres gennem netværket før den øvrige ikke-reserverede trafik. Denne trafikklasse er tiltænkt korte transmissioner, helst enkelte celler, der kan bære kontrolsignaler mellem blokke eller anden systeminformation i netværket. Misbruges Urgent-klassen til større datatransmissioner, kan det få uønskede konsekvenser, idet den relativt korte Urgent-FIFO hurtigt kan blive fyldt, med

deraf følgende ophobning i netværket som resultat.

Er der behov for prioritering af større mængder data, eller større forudsigelighed af latensen, bør en forbindelse oprettes i form af en strøm. Strømme udgør den højest prioriterede trafikklasse, der dog ikke af den grund transmitteres ukritisk. Ved belastning af netværket sker transmissionen i forhold til reservationens omfang.

Det er eksperimentelt blevet eftervist, at der ved større reservation opnås en væsentlig reduktion både i latensen og spredningen af denne.

En svaghed ved det konstruerede netværket er dets sårbarhed overfor ophobning. Ved ophobning sættes prioriteringsmekanismerne i realiteten ud af kraft, da kødannelse blokerer vejen for den prioriterede trafik. Yderligere har ophobning i en switch stort potentiale for at sprede sig i netværket, da køen af celler, der mere eller mindre blokerer et link, også kan reducerer den tilgængelige båndbredde på de foregående link.

Risikoen for ophobning hænger i høj grad sammen med den generelle netværksbelastning, men ved uhensigtsmæssige trafikfordelinger kan den også opstå ved sammenfald af relativt beskedne trafikforløb. Dette sker, når data på flere indgående porte skal videre fra en switch ad samme udgående port, og summen af throughputs overstiger linkkapaciteten. Det er desuden afgørende at sikre, at et passende antal reservationsslots forbeholdes Best-effort trafik. Undlades dette, vil ophobning kunne ske ved relativt beskeden Best-effort trafik, hvilket gør de mange foretagne reservationer nytteløse.

For at undgå disse situationer pålægges systemdesigneren, der forventes at besidde tilstrækkelig viden om det konstruerede systems trafikbehov og -mønstre, at træffe de relevante beslutninger vedrørende fordeling af trafikken. *Source-routing* giver designeren muligheden for at bestemme hvilke links, der skal benyttes til den enkelte kommunikation, og med netværket opbygget i en gitterstruktur, vil der altid findes adskillige mulige veje mellem afsender og modtager i netværket.

Ved større netværk vil den nævnte trafikplanlægning dog let kunne blive uoverskuelig, og gode metoder til udvælgelse af ruter og fordeling af trafikken, anses for at være interessante områder for videre forskning, hvor det forventes, at der kan gøres brug af en del allerede kendt viden fra det grafteoretiske fagområde.

En mulig videreudvikling af den aktuelle netværksswitch, kan ske ved tilføjelse af virtuelle kanaler. Implementering af disse forventes at kunne afhjælpe en del af problematikken omkring ophobning. Virtuelle kanaler tænkes indført med en kanal til hver af de tre trafikklasser, hvilket vil betyde, at trafik ikke vil blive blokeret af ophobede celler fra lavere trafikklasser.

Brugen af virtuelle kanaler blev fra starten afvist med henvisning til det øgede ressourceforbrug. Det er dog muligt, at den øgede kontrol med de forskellige trafikklasser, vil kunne føre til besparelser i den nuværende Reservationsbuffer, hvor bufferkapaciteten, jf. [9] med stor sandsynlighed vil kunne reduceres. Yderligere arbejde indenfor området må vise, hvorvidt dette er tilfældet.

Fastholdes den nuværende arkitektur, er der stadig flere aspekter, der vil være interessante at få undersøgt nærmere. Først og fremmest vil det være nærliggende at få uddybet, hvordan netværket fungerer ved et realistisk trafikscenarie. I den forbindelse vil parametre som FIFO'ernes størrelse, antallet af slots i reservationsbufferen samt den maksimale længde af Urgent-*bursts* kunne justeres, for at få klarlagt deres betydning for trafikkens forløb.

Det væsentlige forhold at få belyst er dog, hvor stort throughput der reelt kan opnås ved en implementering Throughputtet bestemmes af hastigheden hvormed *handshake*-forløb kan gennemføres, samt hvor brede links det vil være praktisk muligt at have mellem netværkets knuder. Det er i litteraturen antydet at mulighederne er meget store, og det vil have endog meget stor indflydelse på valget af arkitektur. Ved meget stor kapacitet, og deraf følgende lav trafikintensitet i netværket, vil relativt beskedne mekanismer være tilstrækkelige til at opnå god servicekvalitet. Ved simuleringerne sås det f.eks. at der var meget lille forskel på latenserne for de forskellige trafikklasser ved 30% baggrundstrafik, og forskellen forventes at være endnu mindre

ved lavere intensiteter.

Tillader de fysiske forhold ikke en væsentlig overdimensionering af netværkets links, vil avancerede reservationsmekanismer som den her beskrevne komme på tale, evt. med yderligere tilføjelser, som det førnævnte eksempel med virtuelle kanaler.

Videre arbejde må belyse hvor tung den beskrevne løsning vil være i en implementering, og det må være disse undersøgelser, der kan danne baggrunden for at vurdere, om det her foreslåede kan have relevans for en praktisk anvendelse.

Foreløbigt synes den valgte løsning at have potentiale til at blive anvendt til konstruktion af on-chip systemer, hvor energi- og arealforbrug er begrænsende faktorer, men der alligevel ønskes mulighed for at kunne differentiere servicen til forskellige typer trafik.

Kapitel 9

Konklusion

Det er i det foregående blevet vist, at det med relativt begrænsede ressourcer er muligt at konstruere et on-chip netværk, der kan tilbyde differentieret service. En service, hvor trafikken prioriteres efter type, og hvor der ved reservation kan opnås en højere grad af forudsigelighed af netværkets latens.

Da netværket tænkes anvendt i GALS systemer, og det derfor ikke er muligt at have global synkronisering, er der i stedet for egentlig *Time-Division-Multiplexing* udarbejdet en tilnærmet løsning baseret på asynkront design.

Der er blevet anslået øvre grænser for latensen af netværkets trafik, og ved simulation er fordelingen af cellelatenser bestemt, ved forskellig belastning af netværket.

Med valget af asynkront design, vil netværkets energiforbrug følge belastningen. Dette resulterer i lavt tomgangsforbrug, og gør netværket attraktivt i forbindelse med systemer, hvor aktiviteten er svingende, eller hvor der forekommer lange hvileperioder.

I det undersøgte netværk er der, som følge af at der kun har kunnet opnås en tilnærmet form for TDM, en vis sårbarhed overfor ophobning. Dette betyder at højt prioriteret trafik, ved høj trafikintensitet, risikerer at blive forsinket på lige fod med lavt prioriteret trafik. Denne uheldige egenskab er et resultat af ønsket om lavt arealforbrug, og forventes at kunne mindskes ved tilføjelse af virtuelle kanaler.

Det er håbet, at dette projekt kan virke som inspiration for den øvrige forskning indenfor området, og eventuelt føre til videre undersøgelse og forbedring af det beskrevne design. I den forbindelse vil bearbejdning af den nuværende beskrivelse til syntetiserbar kode være interessant, da det på den måde vil være muligt at få et mere reelt billede af muligheder og begrænsninger for det beskrevne design.

Litteratur

[1] Functional specification for SystemC 2.0.

[2] SystemC, version 2.0 user's guide.

[3] John Bainbridge and Steve Furber. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on Asynchronous Circuits and Systems*, pages 118–126, March 2001.

[4] John Bainbridge and Steve Furber. Chain: A delay-insensitive chip area interconnect. *IEEE Micro*, 22:16–23, September-October 2002.

[5] Luca Benini and Giovanni De Micheli. Powering networks on chips. *The 14th International Symposium on System Synthesis*, pages 33–38, 2001.

[6] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *Computer*, pages 70–78, January 2002.

[7] Tobias Bjerregaard and Shankar Mahadevan. Modeling asynchronous communication at different levels of abstraction using systemc. *3^{rd} Asynchronous Circuit Design Workshop, ACiD 2003*, 2003.

[8] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture, a Hardware/Software Approach.* Morgan Kaufmann Publishers, inc., 1999.

[9] William J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[10] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. *Design Automation Conference*, pages 684–689, 2001.

[11] K. Goosens, J. van Meerbergen, A. Peeters, and P. Wielage. Networks on sillicon: Combining best-effort and guaranteed services. *Design, Automation and Test in Europe Conference and Exhibition,2002*, pages 423–425, March 2002.

[12] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnetions. *Proceedings of Meeting on Design Automation and Test in Europe*, pages 250–256, March 2000.

[13] Shashi Kumar, Asel Jantsch, Juha-Pekka Soininen, et al. A network on chip architecture and design methodology. *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 117–124, April 2002.

[14] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. Evaluation of the traffic-performance characteristics of system-on-chip communication architectures. *Fourteenth International Conference on VLSI Design, 2001*, pages 29–35, January 2001.

[15] No Name. *SystemC Golden Reference Guide.* Dolous, 2002.

[16] M. Sgroi, A. Mihal, K. Keutzer, et al. Adressing the system-on-a-chip interconnect woes through communication-based design. *Design Automation Conference*, pages 667–672, June 2001.

[17] Jens Sparsø. *Introduction to the Design of Asynchronous Circuits and Systems*. Kluwer Academic Publishers, 2001.

[18] Narayanan Swaminathan and Rabi Mahapatra. Communication architecture synthesis of packet-switched network-on-chip. Technical Report TR-CS-2002-08-0, Texas A&M University, 2002.

Appendix

Bilag A

Ordliste

celle I overensstemmelse med slægtskabet med ATM benævnes pakkerne i dette netværk *celler*. *Celler* har den særlige egenskab at de har fast længde.

celletransmissionstid Tiden det tager at gennemføre et *handshake*-forløb over et link. Er samtidig den tid den ene celle må vente på at blive transmitteret, hvis to celler ankommer samtidig til en tom switch.

effekt Energi pr. tid. Har betydning på fysisk niveau, idet lokal varmeudvikling og energiforsyningen på chippen kan begrænse den tilgængelige effekt.

energi Særligt interessant i forbindelse med batteridrevne anvendelser, hvor den totale energi (ladning), der forbruges er af betydning. Se også *effektforbrug*.

forbindelse En på forhånd reserveret vej for kommunikation mellem sender og modtager.

flit Mindste dataenhed der håndteres ad gangen, kan sammenlignes med et ciffer.

hop Et enkelt delstræk over en kommunikationsvej.

knude Switch, router e.lign., der udgør et knudpunkt i netværksgrafen.

link Forbindelsen mellem to knuder, der er direkte forbundet.

netværksarkitektur Summen af egenskaber, der beskriver et netværk, herunder topologi og rutningsstrategi.

pakke Generel betegnelse for en "klump" data, der transmitteres som én enhed. Kan bestå af en eller flere *flits*.

rutningsstrategi Anvendes både generelt om de valg der har med adressering og valg af rute at gøre. Kan også henvise specifikt til den måde en pakkes vej gennem netværket bestemmes.

source-routing Afsenderen af en pakke bestemmer, hvilken vej denne skal følge gennem netværket for at nå destinationen.

Bilag B

Load i netværk

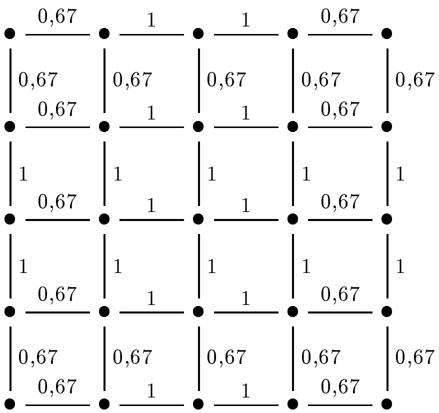
Der er i det beskrevne netværk anvendt *horisontal-first* rutning. Dette valg er ikke baseret på dybere overvejelser, men er truffet udfra et ønske om en simpel rutningsstrategi, hvor der ikke opstår cykliske afhængigheder, og dermed mulighed for *dead-lock* situationer. Der er med *horisontal-first* formentlig ikke truffet det bedste valg mht. fordeling af belastningen i netværket, men algoritmen er enkel og opfylder vores behov for at generere simple ruter til cellerne.

Ved en egentlig implementering kan der, ved valget af rutningsstrategi, med fordel tages højde for de enkelte delblokkes trafikmønstre og krav til netværket. En rutningsstrategi behøver ikke være fælles for alle enheder, så længe det sikres at *dead-lock* ikke opstår som følge af cykliske afhængigheder.

B.1 Loadfordeling ved horisontal-first rutning

For at få et indtryk af trafikfordelingen i et netværk med *horisontal-first* rutning, er der blevet skrevet et simpelt C++ program, der bestemmer belastningen for hvert enkelt link. Belastningen bestemmes udfra antallet af celler, der passerer de enkelte links, hvis der fra samtlige transmittere sendes én celle til samtlige modtagere. Dette giver trafikintensiteten illustreret i figur B.1. Vægten af de enkelte

links angiver belastningen pr. afsendt celle, relativt til de mest belastede links. Af symmetrigrunde bliver værdierne ens i begge retninger på de enkelte links, og der er derfor kun angivet én værdi pr. kant i grafen. I virkelige netværk vil der sjældent forekomme en sådan symmetri, og det vil da være nødvendigt at tage højde for, at linkene består af to modsatrettede unidirektionelle forbindelser.



Figur B.1: Netværksbelastning ved *horisontal-first* rutning

I et 5×5 netværk ses fordelingen af trafikken at være nogenlunde ensartet. Billedet bliver straks noget mere uoverskueligt ved større netværk. Der er dog en generel tendens til, at trafikken ved *horisontal-first* rutning, samles omkring de centrale akser i netværket.

Det ses i figur B.1, at linkene mellem de ni mest centrale knuder er lige hårdt belastet. Dette er grunden til, at testforløbene blev gennemført indenfor dette område. Belastningen her svarer til, at der i gennemsnit transmitteres 1,25 celler for hvert celleinterval. Ved et celleinterval forstås det tidsrum, hvor der afsendes én celle fra hver af samtlige transmittere.

Summen af vægtene i netværksgrafen, når der sammentælles for begge retninger af linkene, vil overstige det afsendte antal celler.

Dette skyldes naturligvis, at hver enkelt celle sendes via flere forskellige links på dens vej til destinationen.

B.2 Baggrundstrafik

Niveauet af baggrundstrafikken defineres som den andel af tiden, hvor der foregår datatransmission på det enkelte link. Da netværket afprøves i den usædvanlige situation, hvor samtlige links i testområdet har samme trafikintensitet, er baggrundstrafikken en entydig betegnelse for, hvor megen øvrig trafik testrafikken skal konkurrere med.

I den anvendte model har vi sat tiden for en transmission over et link til 50 tidsenheder. Dette betragtes som tiden for et fuldt *handshake*-forløb. Afsendes data med et middelinterval på 1000 enheder, som det er tilfældet i de beskrevne testforløb, giver det en belastning af de berørte link på $\frac{50}{1000} \times 100\% = 5\%$.

Afsendes data med samme middelinterval fra samtlige transmittere, vil den forventede belastning af netværkets links være proportional med vægtene i figur B.1 og omvendt proportional med middelintervallet mellem celletransmissioner.

Vi får således flg. sammenhæng:

$$link\_load \propto \frac{link\_vægt}{celleinterval}$$

Idet der ønskes bestemt sammenhængen mellem celleinterval hos transmitterne og trafikintensiteten i de centrale link, opstilles:

$$link\_load = \frac{1,25 \times transmissionstid}{celleinterval} \times 100\%$$

eller

$$celleinterval = \frac{1,25 \times 50}{link\_load} \times 100\%$$

Med dette udtryk findes sammenhængen mellem ønsket baggrundstrafik og det middelinterval for celletransmissioner, der skal benyttes

i transmitterne. Der er ved gennemførsel af testene benyttet nedenstående værdier:

baggrundstrafik	*celleinterval*
30%	208
50%	125
65%	96
70%	89
80%	78
90%	69

Bilag C

Kode

C.1 Simple_packet

Simple_packet.h

```
//  Simple_packet

#ifndef __Simple_packet
#define __Simple_packet

enum TrafficType {CONNECTION_REQUEST,
  CONNECTION_TEARDOWN,
  CONNECTION_SETUP_ACK,
  CONNECTION_SETUP_NACK,
  STREAM_DATA,
  BEST_EFFORT,
  UNDEFINED_PACKET};

class Simple_packet
{
public:
  int forward_path;
  int return_path;
  int connection_ID;
  int sender;
  int time_stamp;
  int data;
  TrafficType traffic_type;

protected:
```



```
 public:

  Simple_packet() : traffic_type(UNDEFINED_PACKET), return_path(0){
  }

  Simple_packet(int path, int sendr, int time_st, int dat = 0,
       TrafficType traf = BEST_EFFORT, int conn_ID = 0) : return_path(0){
    forward_path = path;
    sender = sendr;
    time_stamp = time_st;
    data = dat;
    traffic_type = traf;
    connection_ID = conn_ID;
  }

  bool Simple_packet::isUrgent(){
return (traffic_type == CONNECTION_REQUEST ||
traffic_type == CONNECTION_TEARDOWN ||
traffic_type == CONNECTION_SETUP_ACK ||
traffic_type == CONNECTION_SETUP_NACK);
  }

  bool Simple_packet::isSetup(){
   return traffic_type == CONNECTION_REQUEST;
  }

  bool Simple_packet::isStream(){
return (traffic_type == STREAM_DATA);
  }

  bool Simple_packet::isBE(){
return (traffic_type == BEST_EFFORT);
  }


  bool operator == (const Simple_packet&) const;

  void print( ostream& ) const;
};

inline bool Simple_packet::operator == (const Simple_packet& p) const
{
return (forward_path == p.forward_path &&
return_path == p.return_path &&
sender == p.sender &&
time_stamp == p.time_stamp &&
data == p.data &&
traffic_type == p.traffic_type &&
```

```
connection_ID == p.connection_ID);
}

inline ostream& operator << ( ostream& os, const Simple_packet p)
{
os << "Packet containing: Forward_path: " << p.forward_path
<< ", Return_path: " << p.return_path
<< ", Sender: " << p.sender
<< ", Time_stamp: " << p.time_stamp
<< ", Traffic type: " << p.traffic_type
<< ", Data: " << p.data
<< ", and Connection_ID " << p.connection_ID;
return os;

}

#endif
```

C.2 Abstract_channel

Tilføjelser til oprindelig abstrakt kanal klasse

channel_abstract.h

```
        .
        .
        .

    void peek(T &data);
        .
        .
        .
```

channel_abstract.cpp

```
        .
        .
        .

//Thomas' method to support peeking into the data being sent
template <class T>
void channel_abstract<T>::peek(T& data) {
  // To be used by receiver, to show what data the sender trying to send.
  // Returned value may not be meaningfull if sender is not waiting to send
    data = ch_data;
};
        .
        .
        .
```

C.3 tc_fifo

tc_fifo_ifs.cpp

```
//********************************************************
//********************************************************
//****                                              ****
//**** Extension to sc_fifo_ifs, making it possible to****
//**** peek at fifo-entry at the head of line        ****
//****                                              ****
//****   Author: Thomas Christensen, c971412         ****
//****       DTU, March 2003                         ****
//****                                              ****
//********************************************************
//********************************************************

#ifndef TC_FIFO_IFS_H
#define TC_FIFO_IFS_H

#include "systemc/communication/sc_fifo_ifs.h"


template <class T>
class tc_fifo_in_if
: virtual public sc_fifo_in_if<T>
{
public:

    //blocking peek
    virtual void peek( T& ) = 0;
};


template <class T>
class tc_fifo_out_if
: virtual public sc_fifo_out_if<T>
{
public:

};

#endif
```


tc_fifo.h

Tilføjelser til sc_fifo:

```
        .
        .
        .

    // constructors

    explicit tc_fifo( int size_ = 16 )
: sc_prim_channel( sc_gen_unique_name( "peekable_fifo" ) )
{ init( size_ ); }
        .
        .
        .

    virtual void peek(T&); //blocking peek, waits for data
        .
        .
        .

template <class T>
inline
void
tc_fifo<T>::peek( T& val_ )
{
    while( num_available() == 0 ) {
wait( m_data_written_event );
    }

    if( m_free == m_size ) {
return;
    }
    val_ = m_buf[m_ri];
}
        .
        .
        .
```

C.4 Transmitter

Abstract_transmitter.h

```
// Abstract_transmitter.h: header file

#ifndef __Abstract_transmitter_h
#define __Abstract_transmitter_h

#include <systemc.h>
#include "/home/c971412/Preliminaries/simple_switch/Simple_packet.h"
#include "/home/c971412/CoCentric/Channel/ChannelExpl/channel/ch_send_if.h"

#ifndef SYNTHESIS
#include <ccss_systemc.h>
#endif

static const int nbr_dests = 25;
static const int dests[nbr_dests] = {11,12,13,14,15,
 21,22,23,24,25,
 31,32,33,34,35,
 41,42,43,44,45,
 51,52,53,54,55};


class Abstract_transmitter
: public sc_module
{

public:
    // parameters

    // Value to identify transmitter
    // (use small values to prevent overflow when generating data)
    CCSS_PARAMETER(int, sender_ID);

    // First data value to be sent
    CCSS_PARAMETER(int, start_value);

    // value sent data is incremented with for each transmission
    CCSS_PARAMETER(int, increment);

    // Time for first transmission (in ns) of random data
    CCSS_PARAMETER(int, first_send_NS);

    // Generation scheme: 1 for simple incremental packets, 2 for using file,
    // 3 for random generation,
    // 4 for combined random insertion and as described in file.
    CCSS_PARAMETER(int, Generation_style);
```



```
    // Time interval between transmissions (in ps)
    CCSS_PARAMETER(int, PS_interval);

    // Full path and name of file describing packets to be sent
    CCSS_PARAMETER(char*, path_file);

    // ports
    sc_port<send_if<Simple_packet>, 1> Data_send;

    // initialize parameters
    void InitParameters() {
        int _tmp_sender_ID = 0;
        sender_ID.conditional_init(_tmp_sender_ID);
        int _tmp_start_value = 0;
        start_value.conditional_init(_tmp_start_value);
        int _tmp_increment = 1;
        increment.conditional_init(_tmp_increment);
        int _tmp_first_send_NS = 0;
        first_send_NS.conditional_init(_tmp_first_send_NS);
        int _tmp_Generation_style = 3;
        Generation_style.conditional_init(_tmp_Generation_style);
        int _tmp_PS_interval = 96;
        PS_interval.conditional_init(_tmp_PS_interval);
    }

// default constructor
SC_CTOR(Abstract_transmitter)
{
InitParameters();

number = start_value;
srand(sender_ID); //Seed random function with sender-ID
//Will generate same sequence on each run,
//if randomness between runs is wanted use
//time as argument

// process declarations
SC_THREAD(send_data);
}

~Abstract_transmitter(){
delete data_stream;
}
int number;
ifstream* data_stream;
protected:
void send_data();
void generate_random(Simple_packet&);
```

```
bool read_packet(Simple_packet&);
int choose_dest();
int determine_path(int dest);

}; // end module Abstract_transmitter

#endif
```


Abstract_transmitter.cpp

```
// Abstract_transmitter.cpp: source file
#include "Abstract_transmitter.h"
#include <fstream.h>

void Abstract_transmitter::send_data()
{
switch(Generation_style){
case 1 : // Incremental style
wait(first_send_NS, SC_NS);
while(true){
Simple_packet packet(number
,sender_ID
,sc_clock::time_stamp().value()
,number
//,URGENT
);

Data_send->send(packet);

number += increment; //Prepare for next transmission
wait(PS_interval, SC_PS); //wait untill next transmission

}

break; //unnecessary

case 2 : { // Reading from file
data_stream = new ifstream(path_file);
Simple_packet packet;

while(read_packet(packet)){
//calc. time to transmission
int delay = packet.time_stamp - sc_clock::time_stamp().value();
wait(delay, SC_PS); //wait time as specified in file

packet.sender = sender_ID;

packet.time_stamp = sc_clock::time_stamp().value(); //correct to actual value
Data_send->send(packet);
}
cout << "No more data in file!" << endl;
break;
}

case 3 : { // Random generation. Sends packet at a random point in time,
// within the range:
// [first_send_NS + (pk_nbr)*PS_interval ; first_send_NS + (pk_nbr+1)*PS_interval[
```

```
wait(first_send_NS, SC_NS);

Simple_packet packet;
while(true){
generate_random(packet);

//calc. time to transmission
int delay = packet.time_stamp - sc_clock::time_stamp().value();
wait(delay, SC_PS); //wait until due

Data_send->send(packet);
}
break; //unneeded
}

case 4 : { // Combined generation of data form file and random data

data_stream = new ifstream(path_file);

// prepare packet from file
Simple_packet file_packet;
bool read_packet_waiting;

read_packet_waiting = read_packet(file_packet); //note if data is read

//prepare random packet
Simple_packet random_packet;
generate_random(random_packet);

while(true){
//send one packet when due
if(read_packet_waiting && (file_packet.time_stamp < random_packet.time_stamp)){
//packet read from file should be the next to be sent
//calc. time to transmission
int delay = file_packet.time_stamp - sc_clock::time_stamp().value();
wait(delay, SC_PS); //wait until due

 //update with actual insertion time
file_packet.time_stamp = sc_clock::time_stamp().value();
Data_send->send(file_packet);

read_packet_waiting = read_packet(file_packet); //prepare next (if any)
}
else{
//calc. time to transmission
int delay = random_packet.time_stamp - sc_clock::time_stamp().value();
wait(delay, SC_PS); //wait until due

 //update with actual insertion time
```



```
random_packet.time_stamp = sc_clock::time_stamp().value();
Data_send->send(random_packet);
//prepare next random
generate_random(random_packet);
}
}
break; //unneeded
}

default :
cout << "Invalid Generation_style" << endl;

}
}

int Abstract_transmitter::choose_dest()
{
int dest;
//Selects randomly, based on rand(), between all nodes
do{
//get random index
int index = (int)( ((double) rand()*nbr_dests)/(double)(RAND_MAX) );
dest = dests[index]; //find legal destination in list
}
while(dest == sender_ID); //make sure dest is different from self

return dest;
}

int Abstract_transmitter::determine_path(int dest)
{
if(sender_ID == dest)
cout << "Error: Trying to send to oneself" << endl;
//Method making routing decision - topology specific!
const int grid_width = 5;
const int grid_height = 5;

int x_dist = dest%10 - sender_ID%10; //calculate horisontal distance
int y_dist = dest/10 - sender_ID/10; //calculate vertical distance

//routing scheme: horisontal first

int path;

if(y_dist<0){ //if destination is north of source
path = 12; //last hop is north - extract (1100b)
y_dist++;

while(y_dist<0){
```

```
path = (path << 2) + 0; //left-shift, add north
y_dist++;
}
}
else{
if(y_dist>0){ //if destination is south of source
path = 3; //last hop is south - extract (0011b)
y_dist--;

while(y_dist>0){
path = (path << 2) + 3; //left-shift, add south
y_dist--;
}
}
else{
if(y_dist==0){
if(x_dist<0)
path = /*(path << 2) +*/ 1; //extract packet coming from east
else
path = /*(path << 2) +*/ 2; //extract packet coming from west
}
}// end: if(y_dist>0)
}// end: if(y_dist>0)

while(x_dist<0){
path = (path << 2) + 2; //left-shift, add west
x_dist++;
}
while(x_dist>0){
path = (path << 2) + 1; //left-shift, add east
x_dist--;
}

return path;
}

bool Abstract_transmitter::read_packet(Simple_packet& packet)
{
int type;
if( !(*data_stream >> packet.time_stamp) ) //test if more data in file
return false;

*data_stream >> packet.forward_path;

*data_stream >> type;

*data_stream >> packet.data;

packet.sender = sender_ID;
```



```
switch(type){
case 0 :  packet.traffic_type = CONNECTION_REQUEST;
*data_stream >> packet.connection_ID;
break;

case 1 : packet.traffic_type = CONNECTION_TEARDOWN;
*data_stream >> packet.connection_ID;
break;

case 2 : packet.traffic_type = CONNECTION_SETUP_ACK;
*data_stream >> packet.connection_ID;
break;

case 3 : packet.traffic_type = CONNECTION_SETUP_NACK;
*data_stream >> packet.connection_ID;
break;

case 4 : packet.traffic_type = STREAM_DATA;
*data_stream >> packet.connection_ID;
break;

case 5 : packet.traffic_type = BEST_EFFORT;
break;

default : packet.traffic_type = UNDEFINED_PACKET;
}
return true;
}

void Abstract_transmitter::generate_random(Simple_packet& pack)
{
// Determine random dest
int packet_dest = choose_dest();
pack.forward_path = determine_path(packet_dest);
pack.sender = sender_ID;

int random_offset = (int) ( (double) rand()/ (double)(RAND_MAX) * PS_interval );

// give variance on injection time
pack.time_stamp = first_send_NS*1000 + number*PS_interval + random_offset;

number++;
pack.data = number * increment;
pack.traffic_type = BEST_EFFORT;
pack.connection_ID = 0;
}
```

C.5 Receiver

Abstract_receiver.h

```
// Abstract_receiver.h: header file

#ifndef __Abstract_receiver_h
#define __Abstract_receiver_h

#include <systemc.h>
#include "/home/c971412/Preliminaries/simple_switch/Simple_packet.h"
#include "/home/c971412/CoCentric/Channel/ChannelExpl/channel/ch_receive_if.h"

#ifndef SYNTHESIS
#include <ccss_systemc.h>
#endif


class Abstract_receiver
: public sc_module
{

public:
    // parameters

    // Time for receiver to absorb data (get ready for next reception)
    CCSS_PARAMETER(int, absorb_NS);
    CCSS_PARAMETER(bool, report_reception);
    CCSS_PARAMETER(bool, write_logfile);
    CCSS_PARAMETER(char*, logfile);

    // ports
    sc_port<receive_if<Simple_packet> > Data_receive;

    // initialize parameters
    void InitParameters() {
        int _tmp_absorb_NS = 0;
absorb_NS.conditional_init(_tmp_absorb_NS);
        bool _tmp_report_reception = false;
report_reception.conditional_init(_tmp_report_reception);
        bool _tmp_write_logfile = false;
write_logfile.conditional_init(_tmp_write_logfile);
    }
#ifdef USING_MONITORS
ccss_monitor_msg* msg_monitor;
#endif

// default constructor
SC_CTOR(Abstract_receiver)
```



```
{
InitParameters();

// process declarations
SC_THREAD(receive_data);
}

~Abstract_receiver(){
delete log_data;
}

void receive_data();

ofstream* log_data;
Simple_packet packet_received;
}; // end module Abstract_receiver

#endif
```

Abstract_receiver.cpp

```
// Abstract_receiver.cpp: source file

#include "Abstract_receiver.h"

void Abstract_receiver::receive_data()
{
int cur_time, tof;

if(write_logfile){
log_data = new ofstream(logfile);
if(log_data->is_open()){
*log_data << "Received data at receiver:\t " << name() << endl << endl;
*log_data << "  Time,  Sender,   Sent at,  Traffic Class,  Data,  Time-of-flight";
*log_data << endl;
}
}

while(true){
Data_receive->receive(packet_received);

if(report_reception){
cur_time = sc_clock::time_stamp().value();
//calculate time-of-flight (latency)
tof = cur_time - packet_received.time_stamp;
cout << endl;
cout << "At time: " << cur_time << " receiver: " << name() << " received:\n"
<< packet_received << endl;
cout << "With Time-of-Flight: " << tof << endl << endl;
}

if(write_logfile && log_data->is_open()){
cur_time = sc_clock::time_stamp().value();
//calculate time-of-flight (latency)
tof = cur_time - packet_received.time_stamp;
*log_data << "  " << cur_time;
*log_data << ",   " << packet_received.sender;
*log_data << ",\t     " << packet_received.time_stamp;

if(packet_received.isUrgent())
*log_data << ",\tUrgent";
if(packet_received.isStream())
*log_data << ",\tStream";
if(packet_received.isBE())
*log_data << ",\t   BE";

*log_data << ",\t\t" << packet_received.data;
*log_data << ",\t" << tof;
```



```
*log_data << endl;
}

wait(absorb_NS, SC_NS);

}
}
```

C.6 Simpel switch

Simple_Switch.h

```
// Simple_Switch_Ctrl_tc_FIFO.h: header file

#ifndef __Simple_Switch_Ctrl_tc_FIFO_h
#define __Simple_Switch_Ctrl_tc_FIFO_h

#include <systemc.h>
#include "/home/c971412/Preliminaries/simple_switch/Simple_packet.h"
#include "/home/c971412/CoCentric/Channel/ChannelExpl/channel/channel_abstract.h"
//#include "/home/c971412/Resources/abstract_channel/tc_fifo_ifs.h"
#include "/home/c971412/Resources/abstract_channel/tc_fifo.h"
//#include "/home/c971412/Resources/abstract_channel/tc_fifo_pasted.h"


#ifndef SYNTHESIS
#include <ccss_systemc.h>
#endif

#define CCSS_USE_SC_CTOR
#define VIEW_FIFO_STATE

// forward declarations
class Demux_1_to_4_Ctrl_tc_FIFO;
class Switching_Control;
class Mux_4_to_1_Ctrl_tc_FIFO;

#ifdef CCSS_USE_SC_CTOR
#define CCSS_INIT_CHANNELS_PREFIX :
#undef CCSS_USE_SC_CTOR
#else
#define CCSS_INIT_CHANNELS_PREFIX ,
#endif

#define CCSS_INIT_CHANNELS  CCSS_INIT_CHANNELS_PREFIX \
    In_North("In_North") \
    , Out_North("Out_North") \
    , In_East("In_East") \
    , Out_East("Out_East") \
    , In_South("In_South") \
    , Out_South("Out_South") \
    , In_West("In_West") \
    , Out_West("Out_West") \
    , In_Local("In_Local") \
    , Out_Local("Out_Local") \
    , C41(FIFO_size) \
    , C34(FIFO_size) \
```



```
    , C53(FIFO_size) \
    , C46(FIFO_size) \
    , C48(FIFO_size) \
    , C54(FIFO_size) \
    , C68() \
    , C33(FIFO_size) \
    , C55(FIFO_size) \
    , C72() \
    , C40(FIFO_size) \
    , C47(FIFO_size) \
    , C35(FIFO_size) \
    , C49(FIFO_size) \
    , C42(FIFO_size) \
    , C52(FIFO_size) \
    , C75() \
    , C59("C59") \
    , C65("C65") \
    , C31(FIFO_size) \
    , C39(FIFO_size) \
    , C45(FIFO_size) \
    , C62("C62") \
    , C56(FIFO_size) \
    , C78() \
    , C43(FIFO_size) \
    , C44(FIFO_size) \
    , C63("C63") \
    , C58() \
    , C64("C64")

// Forwards data form input port to relevant output port through internal
// FIFO-queues. Switching decision is based on the two least significant bits
// of the data. Direction is determined as follows:
//
//  00 go North.
//  01 go East.
//  11 go South.
//  10 go West.
//
// If the indicated direction is the way a packet came, this packet is sent
// to the local port.
class Simple_Switch_Ctrl_tc_FIFO// Simple 4-way switch with local connection
: public sc_module
{

public:
    // parameters
    CCSS_PARAMETER(int, FIFO_size);

    // ports
```

```
    sc_port<receive_if<Simple_packet> > In_North;
    sc_port<send_if<Simple_packet> > Out_North;
    sc_port<receive_if<Simple_packet> > In_East;
    sc_port<send_if<Simple_packet> > Out_East;
    sc_port<receive_if<Simple_packet> > In_South;
    sc_port<send_if<Simple_packet> > Out_South;
    sc_port<receive_if<Simple_packet> > In_West;
    sc_port<send_if<Simple_packet> > Out_West;
    sc_port<receive_if<Simple_packet> > In_Local;
    sc_port<send_if<Simple_packet> > Out_Local;

    // channels
    tc_fifo<Simple_packet>  C41;
    tc_fifo<Simple_packet>  C34;
    tc_fifo<Simple_packet>  C53;
    tc_fifo<Simple_packet>  C46;
    tc_fifo<Simple_packet>  C48;
    tc_fifo<Simple_packet>  C54;
    channel_abstract<Simple_packet> C68;
    tc_fifo<Simple_packet>  C33;
    tc_fifo<Simple_packet>  C55;
    channel_abstract<Simple_packet> C72;
    tc_fifo<Simple_packet>  C40;
    tc_fifo<Simple_packet>  C47;
    tc_fifo<Simple_packet>  C35;
    tc_fifo<Simple_packet>  C49;
    tc_fifo<Simple_packet>  C42;
    tc_fifo<Simple_packet>  C52;
    channel_abstract<Simple_packet> C75;
    sc_signal<int> C59;
    sc_signal<int> C65;
    tc_fifo<Simple_packet>  C31;
    tc_fifo<Simple_packet>  C39;
    tc_fifo<Simple_packet>  C45;
    sc_signal<int> C62;
    tc_fifo<Simple_packet>  C56;
    channel_abstract<Simple_packet> C78;
    tc_fifo<Simple_packet>  C43;
    tc_fifo<Simple_packet>  C44;
    sc_signal<int> C63;
    channel_abstract<Simple_packet> C58;
    sc_signal<int> C64;

    // module instances
    Demux_1_to_4_Ctrl_tc_FIFO *M35;
    Demux_1_to_4_Ctrl_tc_FIFO *M31;
    Switching_Control *M24;
    Mux_4_to_1_Ctrl_tc_FIFO *M43;
    Switching_Control *M30;
```



```
    Switching_Control *M25;
    Mux_4_to_1_Ctrl_tc_FIFO *M39;
    Mux_4_to_1_Ctrl_tc_FIFO *M40;
    Mux_4_to_1_Ctrl_tc_FIFO *M42;
    Switching_Control *M28;
    Demux_1_to_4_Ctrl_tc_FIFO *M32;
    Switching_Control *M26;
    Demux_1_to_4_Ctrl_tc_FIFO *M33;
    Demux_1_to_4_Ctrl_tc_FIFO *M34;
    Mux_4_to_1_Ctrl_tc_FIFO *M41;

    // initialize parameters
    void InitParameters() {
        int _tmp_FIFO_size = 1; FIFO_size.conditional_init(_tmp_FIFO_size);
    }
    // create the schematic
    void InitInstances();

    // delete the schematic
    void DeleteInstances();

// default constructor
SC_CTOR(Simple_Switch_Ctrl_tc_FIFO)
  CCSS_INIT_CHANNELS
{
InitParameters();

// process declarations
#ifdef VIEW_FIFO_STATE
SC_THREAD(report_FIFO_load);
#endif

InitInstances();
}

#ifndef SYNTHESIS
// destructor
~Simple_Switch_Ctrl_tc_FIFO()
{
DeleteInstances();
}
#endif

#ifdef VIEW_FIFO_STATE
void report_FIFO_load();
#endif

}; // end module Simple_Switch_Ctrl_tc_FIFO
#undef CCSS_INIT_CHANNELS_PREFIX
```

```
#undef CCSS_INIT_CHANNELS

#endif
```


Simple_Switch.cpp

```
// Simple_Switch_Ctrl_tc_FIFO.cpp: source file

#include "Demux_1_to_4_Ctrl_tc_FIFO.h"
#include "Switching_Control.h"
#include "Mux_4_to_1_Ctrl_tc_FIFO.h"
#include "Simple_Switch_Ctrl_tc_FIFO.h"

void Simple_Switch_Ctrl_tc_FIFO::InitInstances() {
    {int tmp_M35_PS_latency = 100;
ccss_set_parameter("M35.PS_latency", tmp_M35_PS_latency);}
    M35 = new Demux_1_to_4_Ctrl_tc_FIFO("M35");
    {int tmp_M31_PS_latency = 100;
ccss_set_parameter("M31.PS_latency", tmp_M31_PS_latency);}
    M31 = new Demux_1_to_4_Ctrl_tc_FIFO("M31");
    M24 = new Switching_Control("M24");
    {int tmp_M43_PS_latency = 100;
ccss_set_parameter("M43.PS_latency", tmp_M43_PS_latency);}
    M43 = new Mux_4_to_1_Ctrl_tc_FIFO("M43");
    M30 = new Switching_Control("M30");
    M25 = new Switching_Control("M25");
    {int tmp_M39_PS_latency = 100;
ccss_set_parameter("M39.PS_latency", tmp_M39_PS_latency);}
    M39 = new Mux_4_to_1_Ctrl_tc_FIFO("M39");
    {int tmp_M40_PS_latency = 100;
ccss_set_parameter("M40.PS_latency", tmp_M40_PS_latency);}
    M40 = new Mux_4_to_1_Ctrl_tc_FIFO("M40");
    {int tmp_M42_PS_latency = 100;
ccss_set_parameter("M42.PS_latency", tmp_M42_PS_latency);}
    M42 = new Mux_4_to_1_Ctrl_tc_FIFO("M42");
    M28 = new Switching_Control("M28");
    {int tmp_M32_PS_latency = 100;
ccss_set_parameter("M32.PS_latency", tmp_M32_PS_latency);}
    M32 = new Demux_1_to_4_Ctrl_tc_FIFO("M32");
    M26 = new Switching_Control("M26");
    {int tmp_M33_PS_latency = 100;
ccss_set_parameter("M33.PS_latency", tmp_M33_PS_latency);}
    M33 = new Demux_1_to_4_Ctrl_tc_FIFO("M33");
    {int tmp_M34_PS_latency = 100;
ccss_set_parameter("M34.PS_latency", tmp_M34_PS_latency);}
    M34 = new Demux_1_to_4_Ctrl_tc_FIFO("M34");
    {int tmp_M41_PS_latency = 100;
ccss_set_parameter("M41.PS_latency", tmp_M41_PS_latency);}
    M41 = new Mux_4_to_1_Ctrl_tc_FIFO("M41");

    M39->Data_out(Out_West);
    M24->Packet_in(In_West);
    M33->Data_out2(C41);
```

```
    M39->Data_in1(C41);
    M33->Data_out3(C34);
    M42->Data_in0(C34);
    M40->Data_in3(C53);
    M35->Data_out0(C53);
    M42->Data_out(Out_South);
    M40->Data_out(Out_North);
    M43->Data_out(Out_Local);
    M30->Packet_in(In_Local);
    M34->Data_out1(C46);
    M41->Data_in2(C46);
    M32->Data_out0(C48);
    M43->Data_in1(C48);
    M41->Data_in3(C54);
    M35->Data_out1(C54);
    M25->Packet_out(C68);
    M32->Data_in(C68);
    M32->Data_out1(C33);
    M41->Data_in0(C33);
    M42->Data_in3(C55);
    M35->Data_out3(C55);
    M26->Packet_out(C72);
    M33->Data_in(C72);
    M26->Packet_in(In_East);
    M32->Data_out3(C40);
    M42->Data_in1(C40);
    M31->Data_out2(C47);
    M43->Data_in0(C47);
    M34->Data_out2(C35);
    M39->Data_in0(C35);
    M33->Data_out1(C49);
    M43->Data_in2(C49);
    M25->Packet_in(In_North);
    M41->Data_out(Out_East);
    M34->Data_out0(C42);
    M40->Data_in1(C42);
    M34->Data_out3(C52);
    M43->Data_in3(C52);
    M30->Packet_out(C75);
    M35->Data_in(C75);
    M24->Control_signal(C59);
    M31->Control(C59);
    M28->Control_signal(C65);
    M34->Control(C65);
    M31->Data_out0(C31);
    M40->Data_in0(C31);
    M28->Packet_in(In_South);
    M31->Data_out1(C39);
    M41->Data_in1(C39);
```



```
    M33->Data_out0(C45);
    M40->Data_in2(C45);
    M25->Control_signal(C62);
    M32->Control(C62);
    M35->Data_out2(C56);
    M39->Data_in3(C56);
    M28->Packet_out(C78);
    M34->Data_in(C78);
    M31->Data_out3(C43);
    M42->Data_in2(C43);
    M32->Data_out2(C44);
    M39->Data_in2(C44);
    M26->Control_signal(C63);
    M33->Control(C63);
    M24->Packet_out(C58);
    M31->Data_in(C58);
    M30->Control_signal(C64);
    M35->Control(C64);
}

void Simple_Switch_Ctrl_tc_FIFO::DeleteInstances() {
    if (M35) {delete M35; M35 = 0;}
    if (M31) {delete M31; M31 = 0;}
    if (M24) {delete M24; M24 = 0;}
    if (M43) {delete M43; M43 = 0;}
    if (M30) {delete M30; M30 = 0;}
    if (M25) {delete M25; M25 = 0;}
    if (M39) {delete M39; M39 = 0;}
    if (M40) {delete M40; M40 = 0;}
    if (M42) {delete M42; M42 = 0;}
    if (M28) {delete M28; M28 = 0;}
    if (M32) {delete M32; M32 = 0;}
    if (M26) {delete M26; M26 = 0;}
    if (M33) {delete M33; M33 = 0;}
    if (M34) {delete M34; M34 = 0;}
    if (M41) {delete M41; M41 = 0;}
}

#ifdef VIEW_FIFO_STATE
void Simple_Switch_Ctrl_tc_FIFO::report_FIFO_load()
{
while(true){
cout << "Current length of FIFO-queues at node: " << name() << endl;
cout << "From \t to:\tNorth\tEast\tSouth\tWest\tLocal" << endl;
cout << "North\t\t\t --\t\t  " << C33.num_available()
<< "\t\t  " << C40.num_available() << "\t\t  "
<< C44.num_available() << "\t\t  " << C48.num_available() << endl;
cout << "East\t\t\t  " << C45.num_available() << "\t\t --\t\t  "
<< C34.num_available()<< "\t\t  " << C41.num_available()
```

```
<< "\t\t  " << C49.num_available() << endl;
cout << "South\t\t\t  " << C42.num_available()<< "\t\t  "
<< C46.num_available() << "\t\t --\t\t  "
<< C35.num_available() << "\t\t  " << C52.num_available() << endl;
cout << "West\t\t\t  " << C31.num_available() << "\t\t  "
<< C39.num_available() << "\t\t  " << C43.num_available()
<< "\t\t --\t\t  " << C47.num_available() << endl;
cout << "Local\t\t\t  " << C53.num_available() << "\t\t  "
<< C54.num_available() << "\t\t  " << C55.num_available()
<< "\t\t  " << C56.num_available() << "\t\t --" << endl;
cout << endl;

wait(5,SC_NS);
}
}
#endif
```

C.7 Reservation_Switch

Reservation_Switch.h

```
// Reservation_Switch.h: header file

#ifndef __Reservation_Switch_h
#define __Reservation_Switch_h

#include <systemc.h>
#include "/home/c971412/Preliminaries/simple_switch/Simple_packet.h"
#include "/home/c971412/CoCentric/Channel/ChannelExpl/channel/channel_abstract.h"
#include "/home/c971412/Resources/abstract_channel/tc_fifo_ifs.h"
#include "/home/c971412/Resources/abstract_channel/tc_fifo.h"

#ifndef SYNTHESIS
#include <ccss_systemc.h>
#endif

#define CCSS_USE_SC_CTOR
//#define VIEW_FIFO_STATE

// forward declarations
class Demux_1_to_4_Ctrl_tc_FIFO;
class Mux_4_to_1_Ctrl_tc_FIFO;
class Stream_Control;
class Reservation_buffering;
class Switching_Control;

#ifdef CCSS_USE_SC_CTOR
#define CCSS_INIT_CHANNELS_PREFIX :
#undef CCSS_USE_SC_CTOR
#else
#define CCSS_INIT_CHANNELS_PREFIX ,
#endif

#define CCSS_INIT_CHANNELS  CCSS_INIT_CHANNELS_PREFIX \
    In_North("In_North") \
    , Out_North("Out_North") \
    , In_East("In_East") \
    , Out_East("Out_East") \
    , In_South("In_South") \
    , Out_South("Out_South") \
    , In_West("In_West") \
    , Out_West("Out_West") \
    , In_Local("In_Local") \
    , Out_Local("Out_Local") \
    , C49(FIFO_size) \
    , C41(FIFO_size) \
```



```
    , C34(FIFO_size) \
    , C53(FIFO_size) \
    , C91(Extract_fifo_size) \
    , C40(FIFO_size) \
    , C72() \
    , C46(FIFO_size) \
    , C48(FIFO_size) \
    , C54(FIFO_size) \
    , C68() \
    , C33(FIFO_size) \
    , C55(FIFO_size) \
    , C47(FIFO_size) \
    , C35(FIFO_size) \
    , C42(FIFO_size) \
    , C52(FIFO_size) \
    , C75() \
    , C90(Extract_fifo_size) \
    , C59("C59") \
    , C65("C65") \
    , C31(FIFO_size) \
    , C39(FIFO_size) \
    , C89(Extract_fifo_size) \
    , C92(Extract_fifo_size) \
    , C45(FIFO_size) \
    , C62("C62") \
    , C95(Extract_fifo_size) \
    , C56(FIFO_size) \
    , C87(Extract_fifo_size) \
    , C78() \
    , C94(Extract_fifo_size) \
    , C43(FIFO_size) \
    , C44(FIFO_size) \
    , C63("C63") \
    , C86(Extract_fifo_size) \
    , C58() \
    , C64("C64")

// Forwards data form input port to relevant output port through internal
// FIFO-queues. Switching decision is based on the two least significant bits
// of the data. Direction is determined as follows:
//
//  00 go North.
//  01 go East.
//  11 go South.
//  10 go West.
//
// If the indicated direction is the way a packet came, this packet is sent
// to the local port.
class Reservation_Switch// Simple 4-way switch with local connection
```

```
: public sc_module
{

public:
    // parameters
    CCSS_PARAMETER(int, FIFO_size);
    CCSS_PARAMETER(int, Extract_fifo_size);

    // ports
    sc_port<receive_if<Simple_packet> > In_North;
    sc_port<send_if<Simple_packet> > Out_North;
    sc_port<receive_if<Simple_packet> > In_East;
    sc_port<send_if<Simple_packet> > Out_East;
    sc_port<receive_if<Simple_packet> > In_South;
    sc_port<send_if<Simple_packet> > Out_South;
    sc_port<receive_if<Simple_packet> > In_West;
    sc_port<send_if<Simple_packet> > Out_West;
    sc_port<receive_if<Simple_packet> > In_Local;
    sc_port<send_if<Simple_packet> > Out_Local;

    // channels
    tc_fifo<Simple_packet>  C49;
    tc_fifo<Simple_packet>  C41;
    tc_fifo<Simple_packet>  C34;
    tc_fifo<Simple_packet>  C53;
    tc_fifo<Simple_packet>  C91;
    tc_fifo<Simple_packet>  C40;
    channel_abstract<Simple_packet> C72;
    tc_fifo<Simple_packet>  C46;
    tc_fifo<Simple_packet>  C48;
    tc_fifo<Simple_packet>  C54;
    channel_abstract<Simple_packet> C68;
    tc_fifo<Simple_packet>  C33;
    tc_fifo<Simple_packet>  C55;
    tc_fifo<Simple_packet>  C47;
    tc_fifo<Simple_packet>  C35;
    tc_fifo<Simple_packet>  C42;
    tc_fifo<Simple_packet>  C52;
    channel_abstract<Simple_packet> C75;
    tc_fifo<Simple_packet>  C90;
    sc_signal<int> C59;
    sc_signal<int> C65;
    tc_fifo<Simple_packet>  C31;
    tc_fifo<Simple_packet>  C39;
    tc_fifo<Simple_packet>  C89;
    tc_fifo<Simple_packet>  C92;
    tc_fifo<Simple_packet>  C45;
    sc_signal<int> C62;
    tc_fifo<Simple_packet>  C95;
```



```
    tc_fifo<Simple_packet>  C56;
    tc_fifo<Simple_packet>  C87;
    channel_abstract<Simple_packet> C78;
    tc_fifo<Simple_packet>  C94;
    tc_fifo<Simple_packet>  C43;
    tc_fifo<Simple_packet>  C44;
    sc_signal<int> C63;
    tc_fifo<Simple_packet>  C86;
    channel_abstract<Simple_packet> C58;
    sc_signal<int> C64;

    // module instances
    Demux_1_to_4_Ctrl_tc_FIFO *M35;
    Demux_1_to_4_Ctrl_tc_FIFO *M31;
    Mux_4_to_1_Ctrl_tc_FIFO *M43;
    Stream_Control *M62;
    Reservation_buffering *M71;
    Stream_Control *M63;
    Reservation_buffering *M65;
    Reservation_buffering *M70;
    Switching_Control *M30;
    Demux_1_to_4_Ctrl_tc_FIFO *M32;
    Demux_1_to_4_Ctrl_tc_FIFO *M33;
    Stream_Control *M60;
    Stream_Control *M61;
    Demux_1_to_4_Ctrl_tc_FIFO *M34;
    Reservation_buffering *M66;

    // initialize parameters
    void InitParameters() {
        int _tmp_FIFO_size = 10; FIFO_size.conditional_init(_tmp_FIFO_size);
        int _tmp_Extract_fifo_size = 5;
        Extract_fifo_size.conditional_init(_tmp_Extract_fifo_size);
    }
    // create the schematic
    void InitInstances();

    // delete the schematic
    void DeleteInstances();

// default constructor
SC_CTOR(Reservation_Switch)
  CCSS_INIT_CHANNELS
{
InitParameters();

// process declarations
#ifdef VIEW_FIFO_STATE
SC_THREAD(report_FIFO_load);
```

```
#endif

InitInstances();
}

#ifndef SYNTHESIS
// destructor
~Reservation_Switch()
{
DeleteInstances();
}
#endif

#ifdef VIEW_FIFO_STATE
void report_FIFO_load();
#endif

}; // end module Reservation_Switch
#undef CCSS_INIT_CHANNELS_PREFIX
#undef CCSS_INIT_CHANNELS

#endif
```


Reservation_Switch.cpp

```
// Reservation_Switch.cpp: source file

#include "Demux_1_to_4_Ctrl_tc_FIFO.h"
#include "Mux_4_to_1_Ctrl_tc_FIFO.h"
#include "Stream_Control.h"
#include "Reservation_buffering.h"
#include "Switching_Control.h"
#include "Reservation_Switch.h"

void Reservation_Switch::InitInstances() {
    {int tmp_M35_PS_latency = 10;
ccss_set_parameter("M35.PS_latency", tmp_M35_PS_latency);}
    M35 = new Demux_1_to_4_Ctrl_tc_FIFO("M35");
    {int tmp_M31_PS_latency = 10;
ccss_set_parameter("M31.PS_latency", tmp_M31_PS_latency);}
    M31 = new Demux_1_to_4_Ctrl_tc_FIFO("M31");
    {int tmp_M43_PS_latency = 10;
ccss_set_parameter("M43.PS_latency", tmp_M43_PS_latency);}
    M43 = new Mux_4_to_1_Ctrl_tc_FIFO("M43");
    M62 = new Stream_Control("M62");
    {int tmp_M71_Urgent_fifo_size = 3;
ccss_set_parameter("M71.Urgent_fifo_size", tmp_M71_Urgent_fifo_size);}
    M71 = new Reservation_buffering("M71");
    M63 = new Stream_Control("M63");
    {int tmp_M65_Urgent_fifo_size = 3;
ccss_set_parameter("M65.Urgent_fifo_size", tmp_M65_Urgent_fifo_size);}
    M65 = new Reservation_buffering("M65");
    {int tmp_M70_Urgent_fifo_size = 3;
ccss_set_parameter("M70.Urgent_fifo_size", tmp_M70_Urgent_fifo_size);}
    M70 = new Reservation_buffering("M70");
    M30 = new Switching_Control("M30");
    {int tmp_M32_PS_latency = 10;
ccss_set_parameter("M32.PS_latency", tmp_M32_PS_latency);}
    M32 = new Demux_1_to_4_Ctrl_tc_FIFO("M32");
    {int tmp_M33_PS_latency = 10;
ccss_set_parameter("M33.PS_latency", tmp_M33_PS_latency);}
    M33 = new Demux_1_to_4_Ctrl_tc_FIFO("M33");
    M60 = new Stream_Control("M60");
    M61 = new Stream_Control("M61");
    {int tmp_M34_PS_latency = 10;
ccss_set_parameter("M34.PS_latency", tmp_M34_PS_latency);}
    M34 = new Demux_1_to_4_Ctrl_tc_FIFO("M34");
    {int tmp_M66_Urgent_fifo_size = 3;
ccss_set_parameter("M66.Urgent_fifo_size", tmp_M66_Urgent_fifo_size);}
    M66 = new Reservation_buffering("M66");

    M33->Data_out1(C49);
```

```
    M43->Data_in2(C49);
    M62->Packet_in(In_West);
    M33->Data_out2(C41);
    M70->Data_in1(C41);
    M33->Data_out3(C34);
    M66->Data_in0(C34);
    M35->Data_out0(C53);
    M71->Data_in3(C53);
    M70->Out_reinsertion(C91);
    M62->Packet_reinsert(C91);
    M32->Data_out3(C40);
    M66->Data_in1(C40);
    M33->Data_in(C72);
    M60->Packet_out(C72);
    M43->Data_out(Out_Local);
    M30->Packet_in(In_Local);
    M34->Data_out1(C46);
    M65->Data_in2(C46);
    M32->Data_out0(C48);
    M43->Data_in1(C48);
    M35->Data_out1(C54);
    M65->Data_in3(C54);
    M32->Data_in(C68);
    M61->Packet_out(C68);
    M32->Data_out1(C33);
    M65->Data_in0(C33);
    M35->Data_out3(C55);
    M66->Data_in3(C55);
    M71->Data_out(Out_North);
    M43->Data_in0(C47);
    M31->Data_out2(C47);
    M60->Packet_in(In_East);
    M34->Data_out2(C35);
    M70->Data_in0(C35);
    M66->Data_out(Out_South);
    M61->Packet_in(In_North);
    M65->Data_out(Out_East);
    M34->Data_out0(C42);
    M71->Data_in1(C42);
    M34->Data_out3(C52);
    M43->Data_in3(C52);
    M35->Data_in(C75);
    M30->Packet_out(C75);
    M70->Data_out(Out_West);
    M61->Packet_extract(C90);
    M71->Extracted_in(C90);
    M31->Control(C59);
    M62->Control_signal(C59);
    M34->Control(C65);
```



```
    M63->Control_signal(C65);
    M31->Data_out0(C31);
    M71->Data_in0(C31);
    M63->Packet_in(In_South);
    M31->Data_out1(C39);
    M65->Data_in1(C39);
    M61->Packet_reinsert(C89);
    M71->Out_reinsertion(C89);
    M70->Extracted_in(C92);
    M62->Packet_extract(C92);
    M33->Data_out0(C45);
    M71->Data_in2(C45);
    M32->Control(C62);
    M61->Control_signal(C62);
    M66->Out_reinsertion(C95);
    M63->Packet_reinsert(C95);
    M35->Data_out2(C56);
    M70->Data_in3(C56);
    M60->Packet_extract(C87);
    M65->Extracted_in(C87);
    M34->Data_in(C78);
    M63->Packet_out(C78);
    M66->Extracted_in(C94);
    M63->Packet_extract(C94);
    M31->Data_out3(C43);
    M66->Data_in2(C43);
    M32->Data_out2(C44);
    M70->Data_in2(C44);
    M33->Control(C63);
    M60->Control_signal(C63);
    M60->Packet_reinsert(C86);
    M65->Out_reinsertion(C86);
    M31->Data_in(C58);
    M62->Packet_out(C58);
    M35->Control(C64);
    M30->Control_signal(C64);
}

void Reservation_Switch::DeleteInstances() {
    if (M35) {delete M35; M35 = 0;}
    if (M31) {delete M31; M31 = 0;}
    if (M43) {delete M43; M43 = 0;}
    if (M62) {delete M62; M62 = 0;}
    if (M71) {delete M71; M71 = 0;}
    if (M63) {delete M63; M63 = 0;}
    if (M65) {delete M65; M65 = 0;}
    if (M70) {delete M70; M70 = 0;}
    if (M30) {delete M30; M30 = 0;}
    if (M32) {delete M32; M32 = 0;}
```

```
    if (M33) {delete M33; M33 = 0;}
    if (M60) {delete M60; M60 = 0;}
    if (M61) {delete M61; M61 = 0;}
    if (M34) {delete M34; M34 = 0;}
    if (M66) {delete M66; M66 = 0;}
}

#ifdef VIEW_FIFO_STATE
void Reservation_Switch::report_FIFO_load()
{
wait(15,SC_NS);

while(true){
wait(1,SC_NS);
cout << "Current length of FIFO-queues at node: " << name() << endl;
cout << "From \t to:\tNorth\tEast\tSouth\tWest\tLocal" << endl;
cout << "North\t\t\t --\t\t  " << C33.num_available() << "\t\t  "
<< C40.num_available() << "\t\t  " << C44.num_available()
<< "\t\t  " << C48.num_available() << endl;
cout << "East\t\t\t  " << C45.num_available() << "\t\t --\t\t  "
<< C34.num_available()<< "\t\t  " << C41.num_available()
<< "\t\t  " << C49.num_available() << endl;
cout << "South\t\t\t  " << C42.num_available()<< "\t\t  "
<< C46.num_available() << "\t\t --\t\t  " << C35.num_available()
<< "\t\t  " << C52.num_available() << endl;
cout << "West\t\t\t  " << C31.num_available() << "\t\t  "
<< C39.num_available() << "\t\t  " << C43.num_available()
<< "\t\t --\t\t  " << C47.num_available() << endl;
cout << "Local\t\t\t  " << C53.num_available() << "\t\t  "
<< C54.num_available() << "\t\t  " << C55.num_available()
<< "\t\t  " << C56.num_available() << "\t\t --" << endl;
cout << endl;
}
}
#endif
```

C.8 Reservation_buffering

Reservation_buffering.h

```
// Reservation_buffering.h: header file

#ifndef __Reservation_buffering_h
#define __Reservation_buffering_h

#define NBR_SLOTS 32
#define MAX_URGENT_BURST 5
#define NBR_NONRESERVABLE_SLOTS 2

#include <systemc.h>
#include "/home/c971412/Preliminaries/simple_switch/Simple_packet.h"
#include "/home/c971412/CoCentric/Channel/ChannelExpl/channel/ch_receive_if.h"
#include "/home/c971412/CoCentric/Channel/ChannelExpl/channel/ch_send_if.h"
#include "/home/c971412/Resources/abstract_channel/tc_fifo_ports.h"
#include "/home/c971412/Resources/abstract_channel/tc_fifo.h"
#include <ReservationStation.h>

#ifndef SYNTHESIS
#include <ccss_systemc.h>
#endif


// This model describes a reservation unit for a network supporting the
// traffic classes: Urgent, Reserved and Best Effort.
// Reserved trafic is scheduled into the reserved time-slots,
// awaiting transmission.
// Unused slots are filled with data of class best effort.
// Urgent data can surpass best effort traffic, but must not delay
// reserved traffic. Urgent data is therefore kept in a seperate FIFO.
// To let urgent data take the place planned for best-effort and not
// have all following transmissions (and thereby also the resereved traffic)
// delayed, a seperate pointer to the next best-effort packet to send
// is maintained.

// 4 input scheduling unit for network with reserved trafic
class Reservation_buffering
: public sc_module
{

public:
    // parameters

    // Latency measured in ps
    CCSS_PARAMETER(int, PS_latency);
    CCSS_PARAMETER(int, Urgent_fifo_size);
```



```
    // ports
    tc_fifo_in<Simple_packet> Data_in0;
    tc_fifo_in<Simple_packet> Data_in1;
    tc_fifo_in<Simple_packet> Data_in2;
    tc_fifo_in<Simple_packet> Data_in3;
    sc_port<send_if<Simple_packet> > Data_out;
    tc_fifo_in<Simple_packet> Extracted_in;
    tc_fifo_out<Simple_packet> Out_reinsertion;

    // initialize parameters
    void InitParameters() {
        int _tmp_PS_latency = 50;
PS_latency.conditional_init(_tmp_PS_latency);
        int _tmp_Urgent_fifo_size = 3;
Urgent_fifo_size.conditional_init(_tmp_Urgent_fifo_size);
    }

// default constructor
SC_CTOR(Reservation_buffering)
{
InitParameters();
next_turn = 0;
next_input = 0;
next_BE = 0;
urgents_sent = 0;
data_in_buffers = false;
urgent_fifo = new tc_fifo<Simple_packet> (Urgent_fifo_size);
res_stat = new ReservationStation(NBR_SLOTS, NBR_NONRESERVABLE_SLOTS);

for(int i=0; i<NBR_SLOTS; i++){
data_valid[i] = false;
is_stream_data[i] = false;
}

// process declarations
SC_THREAD(receive);

SC_THREAD(transmit);
sensitive << cell_scheduled_event;

SC_THREAD(handle_extracted);

}

~Reservation_buffering(){
delete urgent_fifo;
delete res_stat;
}
```

```
void receive();
void transmit();

protected:
bool data_valid[NBR_SLOTS];
bool is_stream_data[NBR_SLOTS];
//necessary to ensure old BE data is not mistaken for data
//belonging to a new stream
int next_turn; //reservation slot which has next turn
int next_BE; //index of next BE cell waiting
int next_input; //next input to get turn
int urgents_sent;
//number of urgent cells that has been sent while BE-cells were waiting
tc_fifo<Simple_packet>* urgent_fifo;
Simple_packet cell_slots[NBR_SLOTS];
sc_signal<bool> data_in_buffers;
ReservationStation* res_stat; //reservation logic

protected:
void handle_extracted();
bool buffers_empty();
bool schedule_cell(Simple_packet& packet);
bool find_first_available(int& first_avail);
bool find_next_BE(int& next_waiting_BE);
sc_event cell_scheduled_event;

}; // end module Reservation_buffering



#endif
```


Reservation_buffering.cpp

```
// Reservation_buffering.cpp: source file

#include "Reservation_buffering.h"

void Reservation_buffering::receive()
{
while(true){

//Read from the one of the ready inputs, which is next in turn
if(Data_in0->num_available() > 0 || Data_in1->num_available() > 0 ||
Data_in2->num_available() > 0 || Data_in3->num_available() > 0){
Simple_packet tmp_packet;

switch(next_input){
case 0 : if(Data_in0->num_available() > 0){
Data_in0->peek(tmp_packet);
if(tmp_packet.isSetup()){
//make proposed ID unique
tmp_packet.connection_ID = 000 + tmp_packet.connection_ID;
}

//do placement of data
if(schedule_cell(tmp_packet))
//complete transaction by removing from FIFO
Data_in0->read(tmp_packet);
else
wait(sc_get_time_resolution());
}

next_input = 1;
break;

case 1 : if(Data_in1->num_available() > 0){
Data_in1->peek(tmp_packet);
if(tmp_packet.isSetup()){
//make proposed ID unique
tmp_packet.connection_ID = 100 + tmp_packet.connection_ID;
}

//do placement of data
if(schedule_cell(tmp_packet))
//complete transaction by removing from FIFO
Data_in1->read(tmp_packet);
else
wait(sc_get_time_resolution());
}
```

```
next_input = 2;
break;

case 2 : if(Data_in2->num_available() > 0){
Data_in2->peek(tmp_packet);
if(tmp_packet.isSetup()){
 //make proposed ID unique
tmp_packet.connection_ID = 200 + tmp_packet.connection_ID;
}

//do placement of data
if(schedule_cell(tmp_packet))
//complete transaction by removing from FIFO
Data_in2->read(tmp_packet);
else
wait(sc_get_time_resolution());
}

next_input = 3;
break;

case 3 : if(Data_in3->num_available() > 0){
Data_in3->peek(tmp_packet);
if(tmp_packet.isSetup()){
//make proposed ID unique
tmp_packet.connection_ID = 300 + tmp_packet.connection_ID;
}

//do placement of data
if(schedule_cell(tmp_packet))
//complete transaction by removing from FIFO
Data_in3->read(tmp_packet);
else
wait(sc_get_time_resolution());
}

next_input = 0;
break;

default : cout << "Some error occured in receive()'s switch-structure. next_input = "
<< next_input << endl;
}
}//If no data waiting on inputs let time progress before reiterating
else{
wait(sc_get_time_resolution());
}
//Repeat
}
}
```



```
void Reservation_buffering::transmit()
{
while(true){
while(buffers_empty()){//loop is paused if no data is waiting
wait();
}

// if it is time to send from reserved slot and it contains data -send it
if(is_stream_data[next_turn] && data_valid[next_turn]){
wait(PS_latency, SC_PS); //simulate transmission latency
Data_out->send(cell_slots[next_turn]);
data_valid[next_turn] = false;
}
else{ // or send if urgent data is waiting, and not too many have
// been sent since last BE
Simple_packet tmp_cell;
if(urgents_sent < MAX_URGENT_BURST && urgent_fifo->nb_read(tmp_cell)){
//takes advantage of lazy evaluation.
//Second part MUST not be performed unless the cell is being transferred
wait(PS_latency, SC_PS); //simulate transmission latency
Data_out->send(tmp_cell);
urgents_sent++;
}
else{//or send waiting BE, if any
urgents_sent = 0; //new urgent burst allowed if BE had chance to come through
if(find_next_BE(next_BE)){
wait(PS_latency, SC_PS); //simulate transmission latency
Data_out->send(cell_slots[next_BE]);
data_valid[next_BE] = false;
}
else //loop should preferably be left until data becomes ready
wait(sc_get_time_resolution());
}
}
next_turn = (++next_turn)%NBR_SLOTS;
}
}


bool Reservation_buffering::schedule_cell(Simple_packet& packet)
{
if(packet.traffic_type == CONNECTION_REQUEST){
int nbr_cells = packet.data/10; //requested number of slots to be reserved
int freedom = packet.data%10; //indicates the acceptable sceduling variance in
//slot locations (plus-minus) from an even distribution

//get connection_ID used by previous node
```

```
int initial_connection_ID = packet.connection_ID;
//if requested reservation can be fulfilled
if(res_stat->applySetup(nbr_cells, freedom, initial_connection_ID)){
//updata connection_ID with own value
packet.connection_ID = initial_connection_ID;
}
else{//send rejection
packet.forward_path = packet.return_path >> 2;
packet.return_path = 0;
packet.traffic_type = CONNECTION_SETUP_NACK;
packet.connection_ID = initial_connection_ID%100;
Out_reinsertion->write(packet);
return true;
}
}

if(packet.traffic_type == CONNECTION_TEARDOWN){
int connection_ID = packet.connection_ID;
res_stat->teardownConnection(connection_ID);
packet.connection_ID = connection_ID;
}

//put urgent cell in fifo
if(packet.isUrgent()){
if(urgent_fifo->nb_write(packet)){
cell_scheduled_event.notify(SC_ZERO_TIME);
return true;
}
else{
return false;
}
}
// ...or reserved cell in the proper slot
if(packet.isStream()){
//look up the first coming slot for the stream
int next_slot = res_stat->nextSlot(packet.connection_ID, next_turn);

if(next_slot == -1){
cout << "Error no slots reserved for this stream!" << endl;
return false;
}
//make sure slot does not contain data
if(!data_valid[next_slot] ){
//update connection ID
packet.connection_ID = res_stat->getNextHopConnectionID(packet.connection_ID);
cell_slots[next_slot] = packet;
data_valid[next_slot] = true;
is_stream_data[next_slot] = true;
cell_scheduled_event.notify(SC_ZERO_TIME);
```



```
return true; //if successfull
}
else{
//is slot taken (by old BE data), search for other slot (if any)
int search_start = next_slot;
int search_index = search_start;
do{ //look through slots to find one not taken
search_index = res_stat->nextSlot(packet.connection_ID, search_index);
if(!data_valid[search_index]){ //if one is found schedule the cell
data_valid[next_slot] = true;
cell_slots[next_slot] = packet;
is_stream_data[next_slot] = true;
cell_scheduled_event.notify(SC_ZERO_TIME);
return true;
}
} while(search_index != search_start); //continue untill back at starting point

//if no slot could be found
return false;
}
}

//put BE in slot if any available
int first_available;
if(find_first_available(first_available)){
//place at first_available
data_valid[first_available] = true;
cell_slots[first_available] = packet;
is_stream_data[first_available] = false;
cell_scheduled_event.notify(SC_ZERO_TIME);
return true;
}
//not able to schedule cell
return false;
}


bool Reservation_buffering::find_first_available(int& first_avail)
{ //returns true if a slot can be found that has not been reserved and is empty
//index of the slot is written to the variable pointed to by first_avail
int index = next_BE;
for(int i=0; i<NBR_SLOTS; i++){
index = (index+1)%NBR_SLOTS; //let index wrap around
if(!data_valid[index] && res_stat->isAvailable(index)){
first_avail = index;
return true;
}
}
//if no available slot is found return false
```

```
return false;
}


bool Reservation_buffering::find_next_BE(int& next_waiting_BE)
{
// Find next waiting BE-cell, if any
int index = next_waiting_BE;
for(int i=0; i<NBR_SLOTS; i++){
index = (index+1)%NBR_SLOTS; //let index wrap around
if(data_valid[index] && !res_stat->isReserved(index) ){
next_waiting_BE = index;
return true;
}
}
return false; // If no BE-cells are waiting keep next_waiting_BE
}

bool Reservation_buffering::buffers_empty()
{
if(urgent_fifo->num_available()>0){
return false;
}
else{
//bool data_waiting = false;
for(int i=0; i<NBR_SLOTS; i++){
if(data_valid[i])
return false;
}
return true;//!data_waiting;
}
}

void Reservation_buffering::handle_extracted()
{
while(true){
Simple_packet tmp;
if(Extracted_in->num_available() > 0){
Extracted_in->read(tmp);

if(tmp.traffic_type == CONNECTION_SETUP_ACK){
int initial_connection_ID = tmp.connection_ID;
int new_connection_ID = tmp.data;
//confirm setup and update connection_ID for next node
if( res_stat->confirmSetup(initial_connection_ID, new_connection_ID) ){
//reinsert into data stream
tmp.connection_ID = initial_connection_ID%100;
tmp.data = new_connection_ID;//removing indication of input port
Out_reinsertion->write(tmp);
```



```
}
else{
cout << "Error, connection confirmation could not be completed at "
<< name() << endl;
tmp.connection_ID = CONNECTION_SETUP_NACK;
Out_reinsertion->write(tmp);
}
}

if(tmp.traffic_type == CONNECTION_SETUP_NACK){
int initial_connection_ID = tmp.connection_ID;
//cancel setup and update connection_ID for next node
res_stat->cancelSetup(initial_connection_ID);

tmp.connection_ID = initial_connection_ID%100;
//reinsert into data stream
Out_reinsertion->write(tmp);
}

}
else
wait(sc_get_time_resolution());

}
}
```

C.9 Reservationsmekanisme

ReservationStation.h

```
//  ReservationStation

#ifndef __ReservationStation
#define __ReservationStation

#include <iostream.h>
class ReservationStation
{
  public:


  private:
    int* reservation_slots;
    int nbr_slots;
    int slots_reserved;
    int nonreservable_slots;
    int search_start;
    int start_offset;

  ReservationStation(int slots = 16, int nonreserv_slots = 0)
: search_start(0), slots_reserved(0){
   nbr_slots = slots;
nonreservable_slots = nonreserv_slots;
reservation_slots = new int[nbr_slots];
for(int i=0; i<nbr_slots; i++)
   reservation_slots[i] = 0;
  }

  ~ReservationStation(){
delete [] reservation_slots;
  }

  bool isReserved(int slotNbr){
return !(reservation_slots[slotNbr] == 0);
  }

  bool isAvailable(int slotNbr){
   return (reservation_slots[slotNbr] == 0);
  }

  int getNextHopConnectionID(int slot_nbr){
return reservation_slots[slot_nbr];
  }

  bool applySetup(int slots_wanted, int allowable_slack, int& stream_ID){
```



```
if( slots_wanted > slotsAvailable() ) //is it at all possible to reserve slots
   return false;

int search_index;
float ideal_spacing = nbr_slots/(float)slots_wanted;
int slots_found;

for(start_offset=0; start_offset<nbr_slots; start_offset++){
   int slot_index; //for storing a value for hext hop stream_ID
   slots_found = 0;

   for(float i=0; i<nbr_slots && slots_found<slots_wanted; i+=ideal_spacing){
      search_index = (int)((search_start + start_offset + i) +.5);
//search_index is rounded value of ideal slot position
      search_index = search_index%nbr_slots; //let search_index wrap around

      if(reservation_slots[search_index] == 0){
        reservation_slots[search_index] = -1*stream_ID;
//-1 indicates reservation in progress
  slot_index = search_index;
slots_reserved++;
slots_found++;
        continue; //if slot is found continue to next iteration
      }
      else{
        for(int j=1; j<=allowable_slack ; j++){
   int indx = search_index-j;
   if(reservation_slots[indx] == 0 && indx >= 0 && indx < nbr_slots){
     reservation_slots[indx] = -1*stream_ID;
     slot_index = indx;
     slots_reserved++;
     slots_found++;
     break; //ensures only one reservation is made by aborting the for-loop
   }

   indx = search_index+j;
   if(reservation_slots[indx] == 0 && indx >=0 && indx < nbr_slots){
     reservation_slots[indx] = -1*stream_ID;
     slot_index = indx;
     slots_reserved++;
     slots_found++;
     break;
   }
        }//end for
      }//end else

   }
   if(slots_found==slots_wanted){
```

```
     //use some hash-function to change search_offset

     stream_ID = slot_index; //stream_ID to be used for next hop
     return true;
   }

   cleanUp(-1*stream_ID); //clean up before next iteration
}
return false;
  }

  void cancelSetup(int& tmp_strm_ID){
    //clear proper array entries to remove reservation in progress
    int entry = reservation_slots[tmp_strm_ID];
    cleanUp(entry);

    //make entry available for propagation upstream
    tmp_strm_ID = -1*entry;
  }


  bool confirmSetup(int& initial_path_ID, int& new_path_ID){
   bool slots_found = false;
int tmp_ID = -1 * reservation_slots[initial_path_ID];

for(int i=0; i<nbr_slots; i++){
   if(reservation_slots[i] == -1*tmp_ID){
    reservation_slots[i] = new_path_ID;
slots_found = true;
   }
}

//update for transmission to next hop (that is further up stream)
new_path_ID = initial_path_ID;
initial_path_ID = tmp_ID;

   return slots_found;
  }

  void teardownConnection(int& stream_ID){
   if(stream_ID < 0){
  cout << "illegal value used for stream_ID: " << stream_ID << endl;
  return;
}
int old_entry = reservation_slots[stream_ID];

//clear proper entries to remove reservation
cleanUp(old_entry);
stream_ID = old_entry;
```



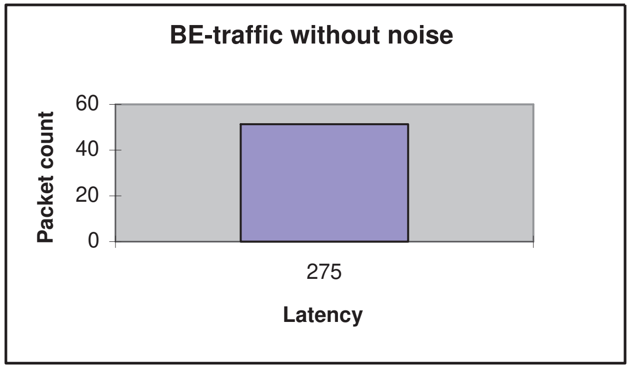
```
  }

  int nextSlot(int stream_ID, int next){
   int next_hop_ID = reservation_slots[stream_ID];
  int index = next;
for(int i=0; i<nbr_slots; i++){
    index = (index+1)%nbr_slots; //let index wrap around
    if(reservation_slots[index] == next_hop_ID)
     return index; //must be ensured, that sender has not passed this index
//while search has been performed
}
//if no reservation exists for this stream ID
return -1;
  }

  int slotsAvailable(){
   return nbr_slots - slots_reserved - nonreservable_slots;
  }

  void printStatus(){
cout << "Reservations:" << endl;

cout << "*---*";
for(int i=0; i<nbr_slots; i++){
   cout << "--*";
}
cout << endl;

cout << "|";
for(int i=0; i<nbr_slots; i++){
   cout << " " << reservation_slots[i] << "|";
}
cout << endl;

cout << "*---*";
for(int i=0; i<nbr_slots; i++){
   cout << "--*";
}
cout << endl;

cout << "slots reserved: " << slots_reserved << endl;
cout << "slots available: " << slotsAvailable() << endl;
  }


  private:
  void cleanUp(int stream_ID){
   if(stream_ID == 0)
  return;
```

```
   for(int i=0; i<nbr_slots; i++){
   if(reservation_slots[i] == stream_ID){
     reservation_slots[i] = 0;
     slots_reserved--;
   }
}
  }
};
#endif
```

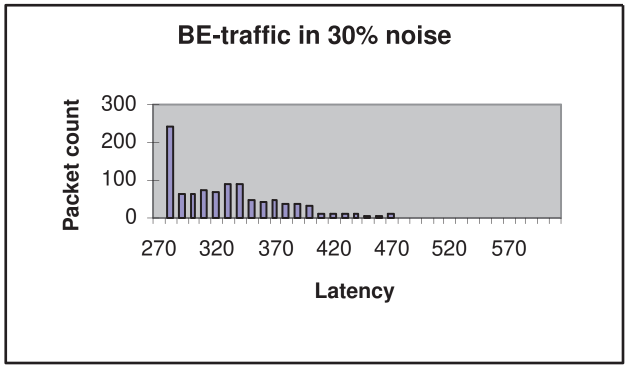
Bilag D

Testdata

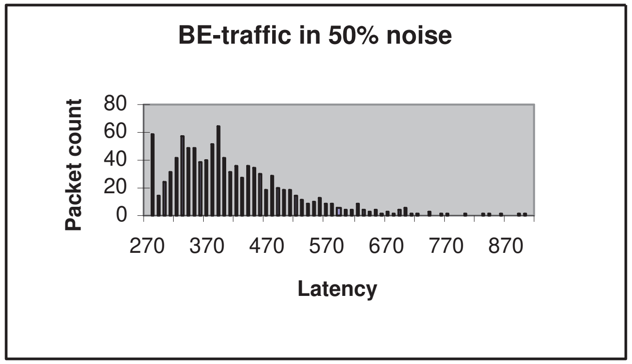
D.1 Best-effort



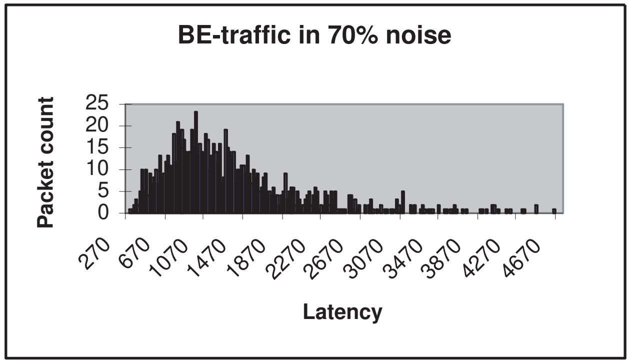
Figur D.1: Best-effort trafik uden baggrundstrafik (kun 50 celler).



Figur D.2: Best-effort trafik med 30% baggrundstrafik.

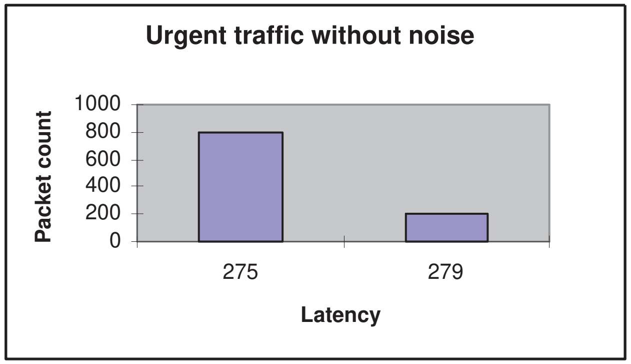


Figur D.3: Best-effort trafik med 50% baggrundstrafik.

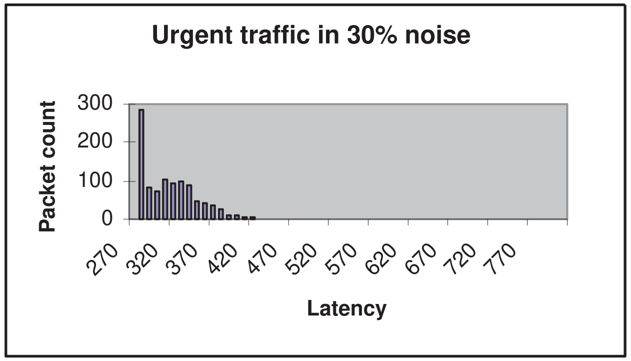


Figur D.4: Best-effort trafik med 70% baggrundstrafik.

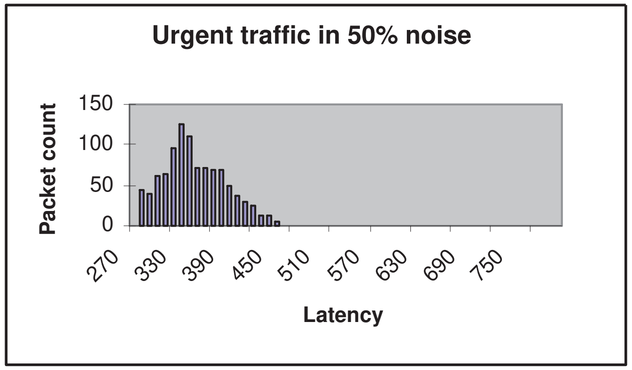
D.2 Urgent trafik



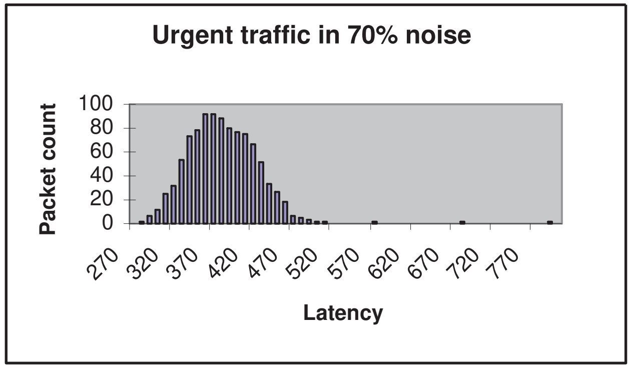
Figur D.5: Urgent trafik uden baggrundstrafik.



Figur D.6: Urgent trafik med 30% baggrundstrafik.



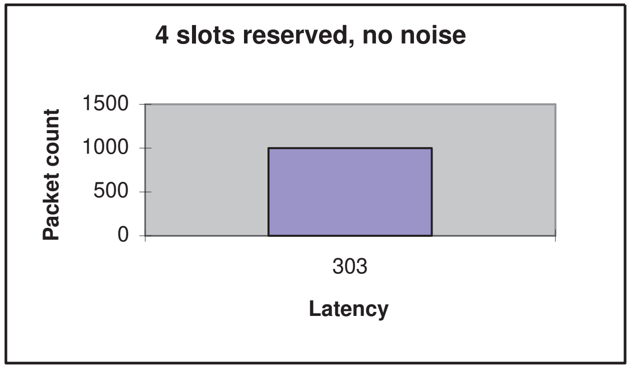
Figur D.7: Urgent trafik med 50% baggrundstrafik.



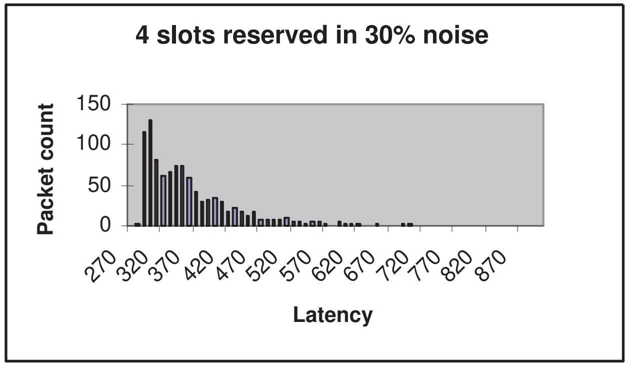
Figur D.8: Urgent trafik med 70% baggrundstrafik.

D.3 Reserveret trafik

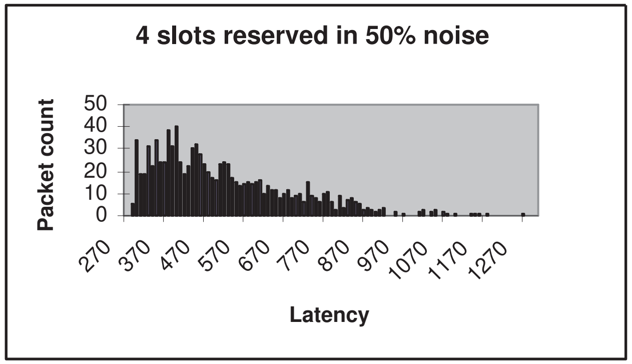
D.3.1 4 reserverede slots



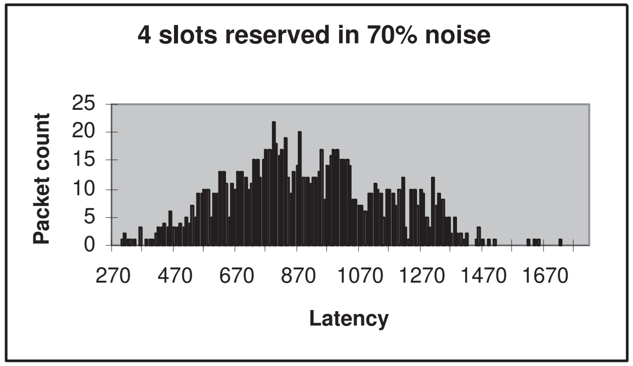
Figur D.9: 4 slots reserveret trafik uden baggrundstrafik.



Figur D.10: 4 slots reserveret trafik med 30% baggrundstrafik.

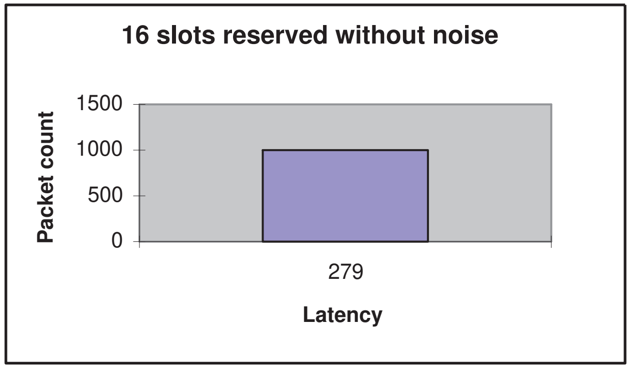


Figur D.11: 4 slots reserveret trafik med 50% baggrundstrafik.

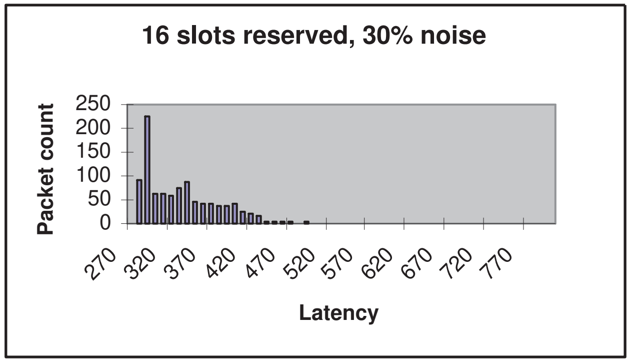


Figur D.12: 4 slots reserveret trafik med 70% baggrundstrafik.

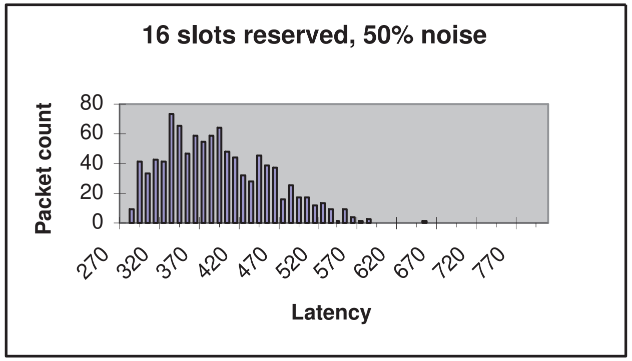
D.3.2 16 reserverede slots



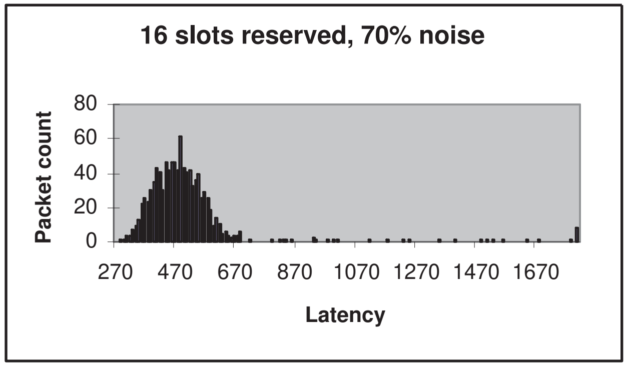
Figur D.13: 16 slots reserveret trafik uden baggrundstrafik.



Figur D.14: 16 slots reserveret trafik med 30% baggrundstrafik.



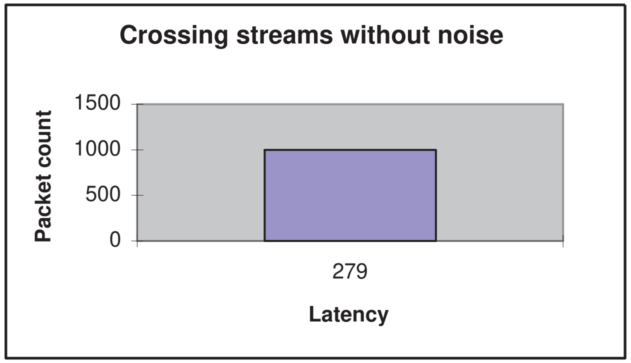
Figur D.15: 16 slots reserveret trafik med 50% baggrundstrafik.



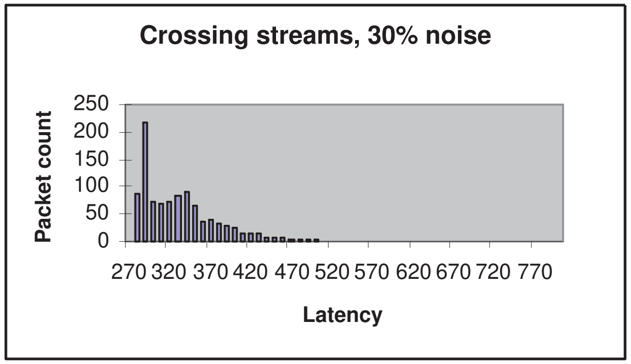
Figur D.16: 16 slots reserveret trafik med 70% baggrundstrafik.

D.4 Flere reservationer

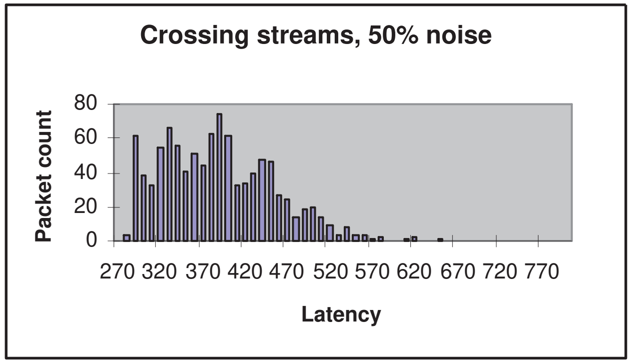
D.4.1 Krydsende strømme



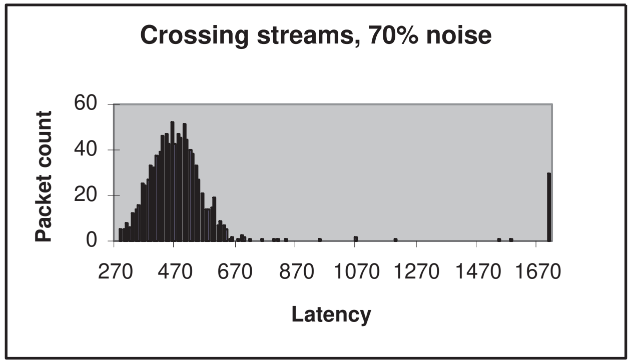
Figur D.17: Krydsende strømme uden baggrundstrafik.



Figur D.18: Krydsende strømme med 30% baggrundstrafik.



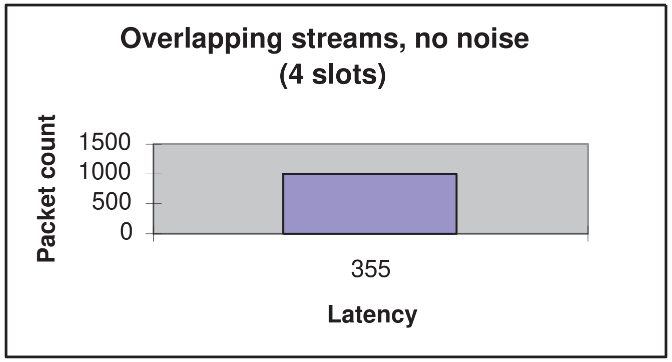
Figur D.19: Krydsende strømme med 50% baggrundstrafik.



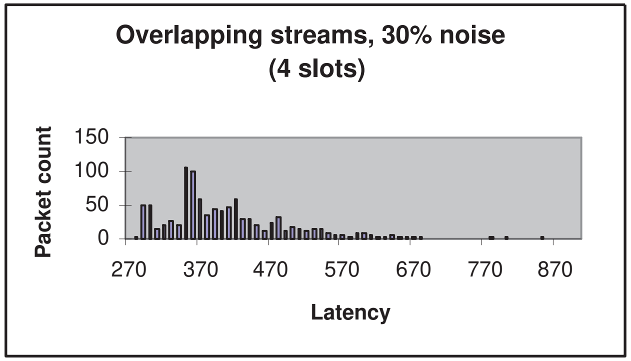
Figur D.20: Krydsende strømme med 70% baggrundstrafik.

D.4.2 Overlappende strømme

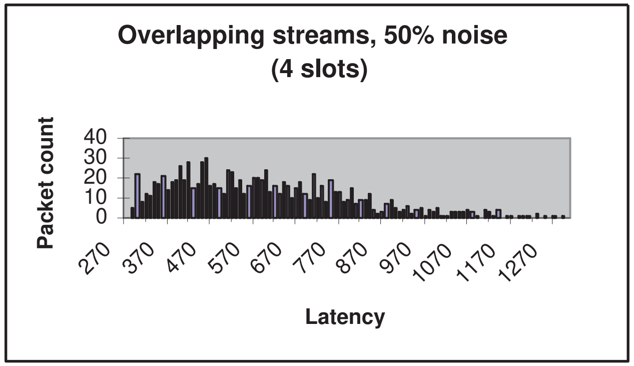
4 reserverede slots



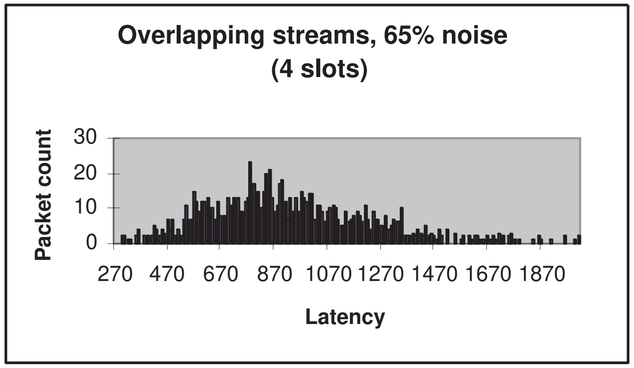
Figur D.21: 4 slot overlappende strøm uden baggrundstrafik.



Figur D.22: 4 slot overlappende strøm med 30% baggrundstrafik.

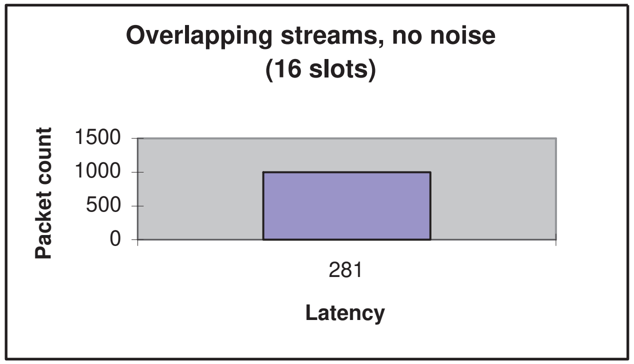


Figur D.23: 4 slot overlappende strøm med 50% baggrundstrafik.

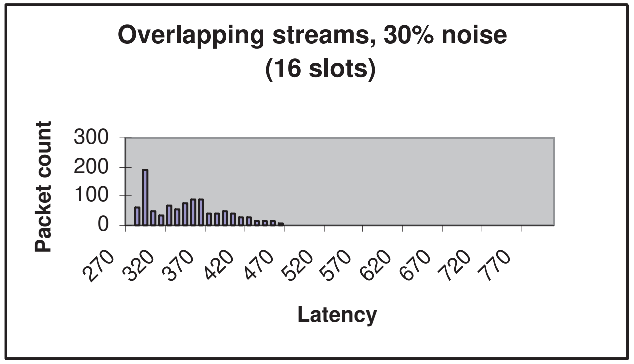


Figur D.24: 4 slot overlappende strøm med 65% baggrundstrafik.

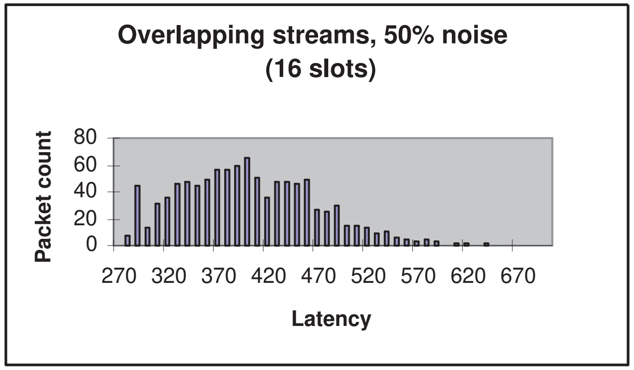
16 reserverede slots



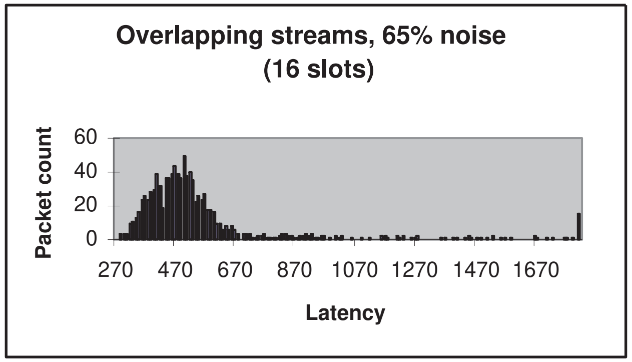
Figur D.25: 16 slot overlappende strøm uden baggrundstrafik.



Figur D.26: 16 slot overlappende strøm med 30% baggrundstrafik.



Figur D.27: 16 slot overlappende strøm med 50% baggrundstrafik.



Figur D.28: 16 slot overlappende strøm med 65% baggrundstrafik.