Aansturing van gesectioneerde on-chip communicatie

Control of Sectioned On-Chip Communication

Kris Heyrman

Promotoren: prof. dr. ir. F. Catthoor, prof. dr. ir. W. Philips, dr. ir. P. Veelaert
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Elektrotechniek

Vakgroep Telecommunicatie en Informatieverwerking
Voorzitter: prof. dr. ir. H. Bruneel
Faculteit Ingenieurswetenschappen
Academiejaar 2009 - 2010

UNIVERSITEIT
GENT

ISBN 978-90-8578-306-0
NUR 959
Wettelijk depot: D/2009/10.500/63

In memory of Hedy Lamarr (1914-2000),
who invented spread-spectrum communication.

"She was not just a pretty face."

Aansturing van gesectioneerde on-chip communicatie

Control of Sectioned On-Chip Communication

Kris Heyrman

There is only one basic way of dealing with complexity: divide and conquer. A problem that can be separated into two sub-problems is more than half solved by that separation. This simple principle can be applied in an amazing variety of ways.

Bjarne Stroustrup, *The C++Programming Language*.

Adventure

```
At End Of Road
You are standing at the end of a road before a small brick
building. Around you is a forest.  A small stream flows out
of the building and down a gully.

>_
```

SUCH an adventure it's been. In June 2004 we stood in Leuven on a parking lot, in front of a large brick building, IMEC's offices. Offering our services for their research effort, they agreed, and we plunged right into the Colossal Cave. In this Cave, one finds horrors and delights. Disorientation strikes easily. Gremlins, bugs and gnomes hide around the corner, then come out and throw spanners in the works. Dwarfs steal research results from the unwary. There are dead ends, deep gorges, cul-de-sacs, low hands and knees passages, complex junctions.

```
> You are in a maze of twisting little passages, all different.
> You are in a little maze of twisty passages, all different.
> You are in a twisting maze of little passages, all different.
>_
```

You will find breath-taking views and campfires, where adventurers huddle at night. Doubt gnawing their insides, they unfold maps, murmur songs that speak of thirst for exploration, of past papers well-received. They will tell you stories about past heroes finding treasure in the Cave and retreating to distant shores. The place may be eerie sometimes, but the environment is international, magazines and fast printers abound, and one can learn a lot. My own quest was intermittent, eventful and at times hair-raising. I would dig for hours, come up for air only once and then, wander on the parking lot, and think about the strange landscape below. In Ghent, in my bathroom, in the morning, I would still muse on the riddles of the Cave.

Today I salute and thank my partners in exploration, whom I met in the Cave: Jin, Miguel, Hua, Florian, Praveen, Murali, Sven, Arnout, Hao. Antonis Papanikolaou, my daily supervisor at IMEC, gradually learned to support my confused explanations and separate the fundamental from the drivel. Donald Knuth stated that premature optimization is the root of all evil. Francky Catthoor provided the philosophical framework and never failed to keep on reading and listening. My colleagues at Hogeschool Gent, department head Marc Vanhaelst and our group's chairman, Jan Beyens, gave great moral support throughout my travails. Lif, my wife, brightened the other hours of my life, and supported one more student in the household. My children Aline and Evert made it worth it. Mieke Geusens and Jan Heyrman, sister-in-law and brother, generously provided food, lodging, stimulating conversation and good company in Leuven. Jan by the way taught me useful model railway layout when I was six. Peter Veelaert showed where algorithms pay off. Pol Marchal explained how to write a scientific paper. Stuart Feldman, a lesser god from the Bell Labs UNIX development group, wrote `make(1)`, and enabled experimentation of the sort done in this work. Wilfried Philips ensured overall sanity. If all was clear, he was never grumpy. Otherwise, he preferred to be clear. William Saphire cleared subtle matters of hyphenation. The examination committee helped improving scope, overall structure and conciseness. St. Christopher suspended traffic jams between Gent and Leuven for the duration.

Special thanks go to people who contributed by publishing their code: Fokko du Cloux from Atlas, Wong Shao Voon from Singapore, Dick Grunwald from University of Colorado at Boulder, A. Dharwadker, and David Eppstein from University of California, Irvine. I am indebted to them, and sending them a complimentary copy of this thesis. I will make sure to publish *my* code in turn.

Now here we are after this long journey, "down there and back up again". It was an ancient initiation ritual. It was the companionship that counted, and the chance I got at mining a diamond.

Kris.

Ghent, October 2009.

```
> You're at end of road again.
> You have achieved the rating: "Experienced Adventurer".
```

Summary

SoCs, a form of very large-scale integration (VLSI) technology, are today prevalent in embedded applications. With the ever-decreasing size of IC features, the International Technology Roadmap for Semiconductors predicts that chips with a billion transistors will be cost-effective soon. Made up from programmable processors, SoC architectures are increasingly seen as the best choice to make use of this complexity, since they leverage established design techniques and existing sets of Intellectual Property (IP). They are simple enough to be designed by teams of limited size, in a time short enough for products to be profitable in the market.

In integrated circuits, latency and chip area were formerly the main cost factors deciding the merit of a product. The deep sub-micron (DSM) domain, the present stage of scaling, causes power consumption to become an extra limitation. Thus power savings are presently studied in all research organizations. In SoCs, power tends to be dissipated by interconnections, rather than by the transistors used for actual information processing. Also, in response to higher demands for bandwidth and reliability, communication architectures are becoming more complex themselves. Interconnection, therefore, is a crucial aspect of future SoC design.

Energy-efficient Sectioned Communication (EESC) is a new methodology for on-chip communication. Some existing techniques, called bus segmentation or splitting, are similar but have a coarser granularity in space and in time. EESC was developed over recent years by IMEC (the Inter-university Micro-electronics Center of Belgium) to combat problems of power consumption by interconnects in the DSM domain. It complements a panoply of other measures taken in the semiconductor industry, operating at various architectural levels. EESC involves both the physical and the upper ("platform") level. Beside focusing on power, it aims to be the best compromise for communication over

medium distances: the wiring inside the tiles that make up a SoC. The high level of activity of information processing inside such tiles makes this type of communication an important contributor to overall costs. Over medium distances, EESC is an improvement over another concept, the network-on-chip (NoC), which is a good compromise for long-distance communication.

Sectioned on-chip communication reduces power consumption by switching off sections of wire at all times when they are not being used. Since communications patterns inside a SoC tile change so fast, to control them is in itself a daunting task. Control of EESC was not studied before: it is the subject of this thesis. We show that such control can originate from the program itself, and re-use decisions made once at compile time by the compiler's scheduler. This avoids unnecessary costs for control circuitry continuously making the same decisions as they recur at run time.

Apart from advocating the principle of programmed control, we establish a novel theory to analyze and design the control circuitry for on-chip networks. If sectioned communication is to be used in the DSM domain and beyond, design engineers must become familiar with our framework and it must be incorporated in automated design software. Also we must convince users that our concepts will work for all topologies and not fail for the large networks of the future. In other words, our approach must be scalable with the number of terminal circuits involved in intra-tile communication.

Our goal is to design a small control plane, one that will be fast enough, and consume little area and power. Optimization for size is our main concern throughout design and analysis. Size of control circuitry relates directly to the size of the state space of the object under control. For a communication network, this space can be very large, unless a description of state can be found that has no redundancy. Our theory establishes a minimal state space description for on-chip networks. When the demands on the communication architecture are well specified, this minimal state space is much smaller than the full combinatorial state space of the network. For a given instruction set architecture of the SoC, the states actually used in operation are called *useful states*. Useful-state space is determined by a process called *Useful-state Analysis (USA)*, focusing on the allocation of switches to paths through the communication architecture. In a sense, the switches are resources and the paths are jobs. USA is a job allocation problem taking account of path concurrency. The result of this analy-

sis relates network control codes to desired states of the network under control. Since the useful-state space is minimal, the encoding, termed useful-state encoding (USE) is minimum-redundancy in a topological sense.

Given these useful-state codes the task of the control plane becomes to convey codes to the network in an optimal way. A design pattern for control circuitry emerges, which we apply to the main domain at IMEC: embedded SoCs for data-intensive applications. The design pattern allows us to distinguish the cost elements of control.

The thesis proposes new methods for the analysis, design and validation of the control plane. It contains examples of key components of the control plane, as well as some original algorithms we developed to design them. The theory of USA and our design framework are used in a number of cases, allowing us to demonstrate that communication architectures can be ranked as to their suitability for EESC by two figures of merit: *Useful-encoding Efficiency (UEE)* and *Intrinsic Sectioning Gain (ISG)*. Also, that two overall use classes must be distinguished: architectures having fixed bandwidth, for instance shared-media, and variable-bandwidth architectures. From the use cases, rules of thumb are deduced, and a conclusion is reached on the scalability of control with many terminals. The thesis ends with an overview of our design framework and a proposal for future work, aiming to either extend the borders of scalability, or the range of communication distance where EESC can be applied.

The work has led to five papers, delivered at three conferences and two symposia, and to a presentation at the System C User's Group, published on the Internet. An overview paper was accepted by the journal *IEEE Transactions on VLSI Systems* and awaits publication.

Samenvatting

In ingebedde toepassingen zijn system-on-chips (SoCs) vandaag de meest toegepaste vorm van VLSI-technologie. Gezien de voortschrijdende miniaturisatie van geïntegreerde circuits, voorspelt de internationale technologie-roadmap voor halfgeleiders (ITRS) dat het binnenkort kostenefficiënt zal zijn om chips met een miljard transistors te produceren. Meer en meer aanziet men SoC architecturen als de beste keuze om gebruik te maken van deze mate van complexiteit. Als samenbouw van programmeerbare processoren laten zij toe gekende ontwerptechnieken en bestaande blokken van Intellectual Property (IP) te herbruiken, terwijl ze toch eenvoudig genoeg blijven om door kleine teams ontworpen te worden, en dit snel genoeg opdat de ontwerpen winstgevend zouden zijn in de markt.

Latentie en oppervlakte op de chip waren tot op heden de kostenfactoren die de kwaliteit van een geïntegreerd circuit het sterkst beïnvloedden. In het diepe submikron (DSM) domein, waarheen de scaleerbaarheid van technologie ons heeft gevoerd, krijgen begrenzingen opgelegd door vermogenverbruik echter steeds meer invloed. Om die reden wordt in alle onderzoeksinstellingen opzoekingswerk gedaan naar vermogensbesparingen. In SoCs wordt vermogen gedissipeerd door draadverbindingen, eerder dan door de transistoren die de informatie verwerken. Omdat hogere eisen worden gesteld aan bandbreedte en betrouwbaarheid, worden anderzijds ook de communicatie-architecturen zelf steeds complexer. Het aspect van de interconnectie is voor toekomstige ontwerpen van SoCs dus cruciaal.

Gesectioneerde energie-efficiënte communicatie (EESC, "Energy-efficient Sectioned Communication") is een nieuwe methode voor on-chip communicatie. Er bestaan andere gelijkaardige technieken, zoals bussegmentatie of bus-splitting, die echter een grovere granulariteit in tijd en ruimte vertonen. EESC werd de laatste jaren ontwikkeld door IMEC, het Interuniversitaire Micro-elektronicacentrum, om ver-

mogensverliezen door interconnecties in het DSM domein te bestrijden. EESC is complementair aan andere maatregelen die overal in de halfgeleiderindustrie worden genomen en die werken op verscheidene architecturale niveaus. EESC zelf is werkzaam op fysisch zowel als op een hoger niveau, namelijk dat van het platform. De methode beoogt niet alleen vermogensbesparing, maar wil ook het beste compromis uitmaken voor communicatie op middellange afstand: dit wil zeggen, voor bedrading binnenin de tegels van een SoC. De activiteitsgraad van de informatieverwerking binnen deze tegels ligt hoog. Dat betekent dat de bedrading een belangrijke bijdrage levert tot de totale kost. EESC is, over middellange afstand, beter dan een ander concept, network-on-chips (NoCs), welke dan weer voor lange afstanden, tussen verschillende tegels, een goed compromis vormen.

Gesectioneerde on-chip communicatie bespaart vermogen door de secties van draden op alle ogenblikken dat zij niet in gebruik zijn af te schakelen. Het is een uitdagende taak de communicatiepatronen binnen de tegel van een SoC aan te sturen, omdat de patronen zo snel veranderen. Tot op heden was er nog geen studie van de aansturing van EESC ondernomen: het werd het onderwerp van dit doctoraat. Wij tonen aan dat de aansturing kan voortkomen uit het programma zelf, en hergebruik maken van beslissingen door de scheduler bij het compileren. Dit vermijdt het oplopen van weerkerende kosten voor besturingscircuits die voortdurend at run time dezelfde beslissingen maken.

Los van dit principe van aansturing door het programma, stellen we in dit proefschrift een oorspronkelijke raamwerk, een theorie, voorop voor het analyseren en ontwerpen van de sturing van on-chip netwerken. Willen we gesectioneerde communicatie invoeren in het DSM domein en nog verder, dan moeten ontwerpingenieurs vertrouwd raken met ons raamwerk en dient het opgenomen te worden in software voor geautomatiseerd ontwerp. Evenzo moeten gebruikers ervan overtuigd worden dat onze concepten blijven werken voor alle topologieën, en voor de grote netwerken die in de toekomst kunnen worden verwacht. Met andere woorden moet onze benadering scaleerbaar zijn met het aantal terminals dat binnen een tegel aan de communicatie deelneemt.

Ons opzet is een klein aansturingsvlak te maken, dat snel genoeg zal zijn en weinig oppervlakte of vermogen gebruiken. Onze belangrijkste zorg doorheen ontwerp en analyse is optimalisatie voor kleine afmetingen. De afmetingen van een sturingscircuit staan in rechtstreeks verband met die van de toestandsruimte van het object dat wordt bestuurd. Bij een communicatienetwerk kan zulke toestandsruimte erg groot

zijn, tenzij we er in slagen een redundantie-vrije beschrijving van de toestand te vinden. Onze theorie stelt een minimale beschrijving van de toestandsruimte van on-chip netwerken voorop. Worden de vereisten gesteld aan de communicatie-architectuur correct geformuleerd, dan is deze toestandsruimte kleiner dan de complete combinatorische toestandsruimte van het netwerk. Voor een gegeven instructieset-architectuur van de SoC noemen wij de toestanden die effectief in werking gebruikt worden, de *nuttige toestanden*. De nuttige toestandsruimte kan worden vastgesteld door *nuttige-toestandsanalyse (USA, "Useful-state Analysis")*. Deze procedure spitst zich toe op het toewijzen van schakelaars aan paden doorheen de communicatie-architectuur. In zekere zin zijn de schakelaars resources en de paden jobs: USA is een probleem van job-allocation, dat rekening houdt met het gelijktijdig bestaan van paden. Het eindproduct van dit vraagstuk, een *padverzamelingstabel* genaamd (PSLT, "path-set lookup table"), brengt codes in verband met gewenste toestanden van het gestuurde netwerk. Omdat de nuttige toestandsruimte minimaal is, is de resulterende codering, die we *nuttige toestandscodering (USE, "useful-state encoding")* noemen, minimaal redundant in topologische zin.

Gegeven deze aanstuurcodes wordt de opdracht van het aansturingsvlak de codes op optimale wijze over te brengen naar de schakelaars. Er ontstaat een ontwerppatroon voor de sturingscircuits, dat we toepassen op het belangrijkste domein van IMEC: embedded SoCs voor data-intensieve toepassingen. Het ontwerppatroon laat toe alle elementen van de kosten van aansturing te onderscheiden.

Het proefschrift stelt verder een nieuwe aanpak voor van de analyse, het ontwerp en de validatie van het aansturingsvlak. Naast enige oorspronkelijke algoritmes, ontwikkeld om essentiële onderdelen te kunnen ontwerpen, bevat het voorbeelden van die onderdelen. De USA-theorie en het raamwerk voor ontwerp worden gebruikt in een aantal toepassingen. Dit laat ons toe aan te tonen dat communicatie-architecturen kunnen gerangschikt worden met betrekking tot hun geschiktheid voor EESC, door middel van twee indicatoren: *rendement van nuttige codering* (UEE, "useful-encoding efficiency") en *intrinsieke winst door sectionering* (ISG, "Intrinsic Sectioning Gain"). Verder kan een onderscheid gemaakt worden tussen twee gevallen van gebruik: architecturen met vaste bandbreedte, bijvoorbeeld gedeelde media, en architecturen met variabele bandbreedte . Uit deze toepassingen kunnen we vuistregels afleiden, naast een besluit over de scaleerbaarheid met vele terminals. Het proefschrift eindigt met een overzicht van ons raamw-

erk voor ontwerp en een voorstel voor verder onderzoek. Dit laatste zou zich kunnen richten op twee onderscheidbare gebieden: enerzijds zouden de grenzen van de scaleerbaarheid kunnen verlegd worden, of anderzijds zou het afstandsbereik van de communicatie waarover men EESC kan toepassen kunnen worden uitgebreid.

In het kader van dit werk werden vijf artikels gepubliceerd, voorgebracht op drie conferenties en twee symposia, en een presentatie gegeven voor de System C User's Group en gepubliceerd op het Internet. Een overzichtsartikel werd geaccepteerd door het tijdschrift *IEEE Transactions on very large-scale integration (VLSI) Systems* en wacht op publicatie.

Acronyms

6W3T 6-way, 3-terminal

ALU arithmetic-logic unit

AHB Advanced High-performance Bus

AMBA Advanced Microcontroller Bus Architecture

APM Advanced Peripheral Bus

APS all-paths set

ARM Advanced RISC Machines

ASB Advanced System Bus

ASIC application-specific integrated circuit

BFSIT breadth-first search iterator

BIT Beckett iterator

CA communication architecture (A_C)

CDFG control/data flow graph

CP communication processor

CPS concurrent path-set

CMP chip-level multiprocessor

CMT chip multi-threading

CMOS complementary metal-oxide semiconductor

CRISP coarse-grain reconfigurable instruction-set computer

CTS concurrent transfer-set

DAB Digital Audio Broadcasting

DIS Dharwadker's independent set algorithm

DMA direct memory access

DRAM dynamic random-access memory

DSM deep sub-micron

DSP digital signal processor

DTSE Data Access and Storage Management for Embedded Programmable Processors

EESC Energy-efficient Sectioned Communication

FAR find all routes algorithm

FDR find distinct routes algorithm

FPGA field-programmable gate array

FIFO first-in first-out

FFT fast Fourier transform

FO4 fan-out 4

FSM finite-state machine

FU functional unit

GSM Global System for Mobile Telecommunication

HPC high-performance computing

IC integrated circuit

IMEC Inter-university Micro-electronics Center

ILP integer (linear) programming

IP Intellectual Property

IPv6 Internet Protocol

IPS independent path-set algorithm

IS instruction set

ISA instruction set architecture

ISG Intrinsic Sectioning Gain (G_{IS} or G'_{IS} (schedule-neutral))

ITRS International Technology Roadmap for Semiconductors

HDL Hardware Description Language

L1 level-1

LESB linear energy-efficient sectioned bus

LNDL Little Network Description Language (*Elendil*)

LP linear programming

MIP mixed integer programming

MIPS maximal independent path-set algorithm

MPU microprocessor unit

NoC network-on-chip

NUMA non-uniform memory access

PAG path allocation graph (G_{PA})

PGW Power Gating for Wires

PIT powerset iterator

PPS path-powerset algorithm

PRAG path-resource allocation graph (G_{PRA})

PSL Property Syntax Language

PSLT path-set lookup table

QoS quality of service

RTL register-transfer level

RAH resource allocation hypergraph (H_{RA})

SCPS set of concurrent path-sets (S_{CPS})

SBA Segmented Bus Analysis

SoC system-on-chip

SOC sequence of combinations

SRAM static random-access memory

STL Standard Template Library

STM ST Microelectronics

SMP symmetric multi-processing

TA terminal arrangement

TAD Technology-aware Design

TCH transfer-compatibility hypergraph

TSLT transfer-set lookup table

TL transaction-level

TLB transport loop buffering

TSU Terminal Select Unit

TTA Transport-triggered Architecture

UEE Useful-encoding Efficiency (η_{UE})

USA Useful-state Analysis

USE useful-state encoding

VLIW very-long-instruction word

VLSI very large-scale integration

UPS useful-paths set

UML Unified Modeling Language

XML Extended Markup Language

Contents

Figures

Tables

Chapter 1

Introduction

We're like penguins with the ice melting around them.
We keep on doing the same things.

G. Daniel Hutcheson, *on power consumption by interconnects.*

In this introductory chapter we situate the research presented in this PhD thesis. We describe the problem of power consumed by on-chip wire interconnection; a methodology, EESC, intended to address this problem; the related problem of interconnect reliability degradation; and the importance of control in EESC.

We will relate our research to the program of IMEC's technology-aware design (TAD) group, highlight our own contributions and those of others in the group, and finally preview the material to come.

In the deep sub-micron (DSM) domain, wires dissipate more power than transistors. Energy-efficient Sectioned Communication (EESC)[1] is a form of bus segmentation that reduces the power loss from on-chip interconnects, by switching off the supply voltage from inactive drivers, instruction-cycle by instruction-cycle. How EESC must be controlled, however, was not clear at the beginning of this research. It was anticipated that control strategies could easily run into scalability problems for large or complex networks. Our work intends to provide

[1] Another term for the same concept, Power Gating for Wires (PGW), is more concise, indicating at the same time how it is done, and stressing the aspect of control. It was used in [62]. In this work, we do not use it. The term EESC, describing the design methodology by its intention, is more common at TAD.

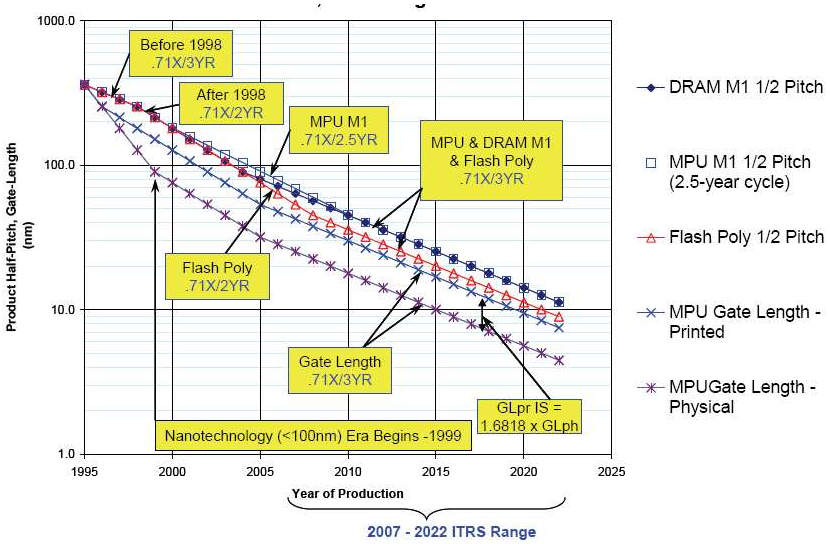


Figure 1.1: Trends in feature length. A feature, which characterizes a technology node, is the physical or printed length of an MPU gate, or the half-pitch of a memory cell. (Source: ITRS)

a framework for optimal control plane design, and to study and improve the limits that scalability imposes. To do so, we first explain the urgency of the problem of excessive power consumption in the DSM, its particular importance for wire interconnects within system-on-chips, and the nature of the (initial) misgivings about its control.

1.1 On-chip Communication and EESC

1.1.1 Energy Consumption in the Deep Sub-micron Domain

In the semiconductor industry, miniaturization is expressed in terms of lengths of features that characterize a generation of technology (a "node"). Driven by economic opportunity, and guided by the International Technology Roadmap for Semiconductors (ITRS) [67], the feature lengths of new technology nodes have decreased over the last forty years at a rate of approximately 0.71 per 2.5 year, as shown in Fig. 1.1. It is the intention of the industry to continue this advance. Between individual market segments, the front of this advance is uneven, depending on the prevailing market dynamics. At present,

technologies for cutting-edge processor cores and high-density static random-access memory (SRAM) employ 32 and 45 nm nodes. Recent field-programmable gate arrays (FPGAs) (Stratix IV, Virtex 5). achieve feature lengths of 40 and 65 nm. New system-on-chips (SoCs) and application-specific integrated circuits (ASICs), which may be low-power, embedded, or small-volume, lag somewhat in the adoption of new nodes. These integrated circuits (ICs) are being implemented in a range from 70 nm up to 360 nm, and more. Some high-volume markets for low-power consumer SoCs, like mobile telephony digital baseband receivers, are presently reaching 45 nm nodes in production [21]. Research efforts, of course, have always anticipated the introduction of nodes by a number of years.

For the ITRS plan, excessive energy consumption is a relatively new and growing problem. This was illustrated by the emergence in 2004 of a power wall in CMOS microprocessor design, forcing a major revision on the roadmaps for main-line microprocessing. According to Flynn [35], the essence of the power-wall problem is the cubic trade-off between time and power, where $T^3 \times P$ is constant over technology. This forces designers to use relatively cheap area to offset the expensive power required by high clock rates. The problem is in no way limited to the microprocessor market.

Horowitz [65] formulates the trade-off in the following way: one can *(i)* trade speed for power, either by *voltage scaling*, or *transistor sizing*. This does not by itself decrease power consumption. *Technology scaling* does save power for local wires and gates, but is hampered by the fact that the threshold voltage for CMOS devices does not scale along with the supply voltage. This introduces new power losses by leakage. *(ii)* Additionally, one can make sure that *idle circuitry does not use any power*, either from static losses or from unnecessary transitions. For this, it must be possible to switch circuits off selectively. Systems must be divided in a heterogeneous fashion, so that in some phases of operation the functionality of some blocks can be deactivated. According to Horowitz, the most significant power reductions using this strategy can be obtained by examining the problem at the system level. *(iii)* One can also exploit *parallelism* where it is available at small enough cost. *(iv)* With creative insight, one can implement the computation tasks in *a more efficient way*.

Since strategy *(i)* is in the realm of physical design, and strategy *(iv)* is for the algorithm designer or software architect, the hardware

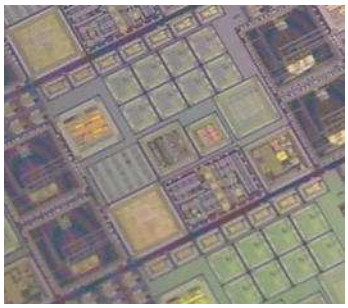


Figure 1.2: Photograph of a tile-based SoC.

system designer's workspace is defined by the axes of strategies *(ii)-(iii)*: heterogeneity and parallelism, at different hierarchical levels, with all granularities suitable to the design. Those two approaches offer the best opportunities to address the energy consumption problem, as we encounter it in the design of intra-tile communication for SoCs.

1.1.2 Intra-tile Communication

The worsening of the trade-offs from technological progress, and the strategies employed to rebalance them, are common to all fields of application, including the field of *programmable embedded SoCs* (Fig. 1.2), which have been our concern at the Inter-university Micro-electronics Center (IMEC). SoCs are combinations of heterogeneous hardware blocks, operating in parallel to a high degree. The blocks are managed by sophisticated microprocessors, themselves running under control of dedicated software. We call the parts of SoCs under control of a single program a "tile".

From the above, we see why the problem of power consumption places an extra burden on the intra-tile communication architecture. A typical intra-tile network is shown in Fig. 1.3. Because of the demands of heterogeneity and maximal parallelism, the data rate to be transferred within a tile is high. The internal communication network should provide large enough bandwidth at low power. This must be achieved using the same principles cited above: heterogeneity, and parallelism in communication.

For interconnects a specific physical scaling problem exists. The delay of global wires scales badly in proportion to the delay of local wires and gates, as is shown in Fig. 1.4. This implies an increase in line

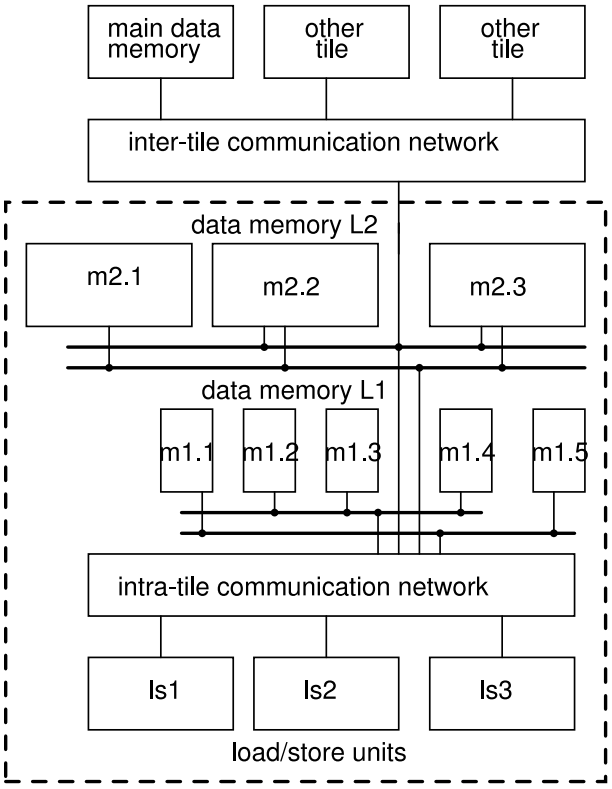


Figure 1.3: The synchronous network inside a processor tile (within the dashed box) connects memories and functional units internal to a tile.

driver power for medium and long-distance communication. Consequently, low-power interconnects have in recent years become a major challenge to the industry. Designers have been forced to rethink many aspects of their craft, as seen from a number of surveys on this matter [23, 26, 37, 53, 54, 75, 105]. Many remedies are already applied today [7, 12, 98]. To be effective, such a remedy must approach the problem at once at diverse levels: structural (like, for instance, 3D chips), modeling (diagonal wires), procedural (fault tolerant communication), circuit-level (low voltage signaling), architectural (communication architecture selection, bus isolation), and system-level (power management, voltage scaling). None of these remedies are particular to a spe-

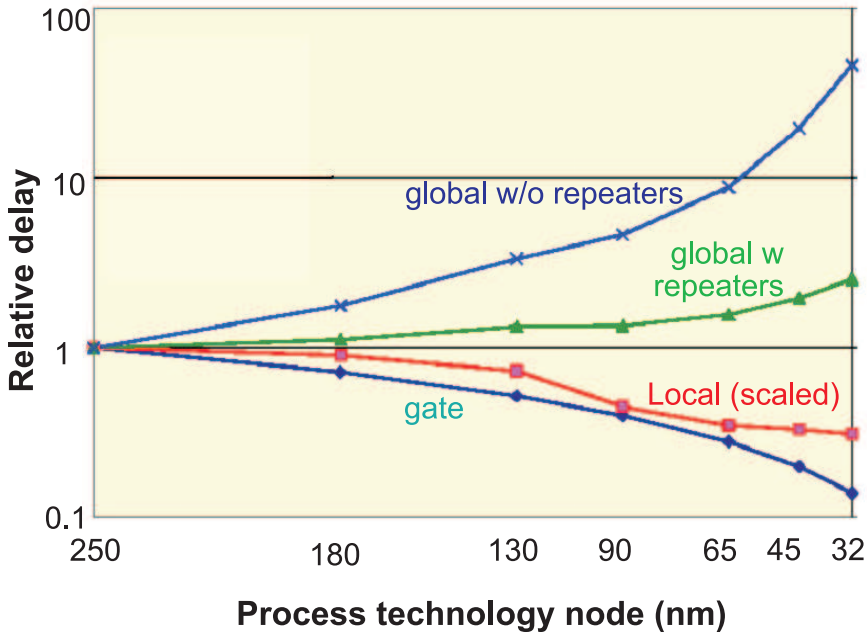


Figure 1.4: Scaling properties of global wire delays vs. delay of local wires and gates. (Source: ITRS)

cific technology node or application field. Taken together, they form an array of related design techniques developed by the semiconductor industry to address the problem of low-power on-chip communication.

Interconnect reliability degradation in SoCs An additional problem in SoC interconnects, touched upon in this thesis, is again caused by technology scaling. Although tremendous improvements in chip reliability have been achieved, complementary metal-oxide semiconductor (CMOS) materials have been pushed to their limits. Due to the bad scaling characteristics of interconnects, illustrated in Fig. 1.4, copper which has better RC properties now replaces aluminum. Use of copper wires, and their insulators which typically use low-κ dielectrics, expose reliability problems that will be exacerbated by scaling. The physical phenomena causing these are called Electron Migration (EM) and Time-Dependent Dielectric Breakdown (TDDB) [52]. These factors of reliability degradation introduce a "wear-out" mechanism into scenarios of failure to be taken in account. Thus special measures must be taken to ensure survivability of on-chip communication in future

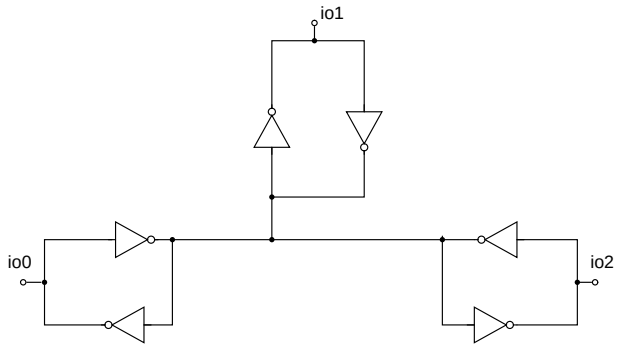


Figure 1.5: A six-way three-terminal (6W3T) wire sectioning switch.

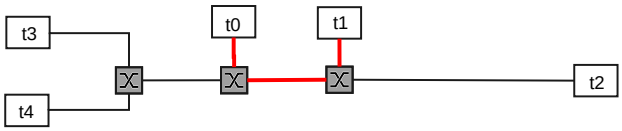


Figure 1.6: Principle of wire sectioning by switches.

technology nodes. One such possible measure is the re-routing of communication paths over the lifetime of a chip.

1.1.3 Energy-efficient Sectioned Communication

EESC is a promising approach to remedy the problem of energy consumption by interconnects. In EESC, we exploit deactivation of subcircuits in the form of wire sectioning. The physical basis of EESC is heterogeneity: the partitioning of similar resources in parts that can be controlled separately. It divides wires in sections and refrains from driving the wire sections during cycles when this is not necessary for the purpose of information processing. The sectioning is performed by switches that incorporate line drivers. A typical such switch is shown in Fig. 1.5: it is a six-way three-terminal (6W3T) switch. It contains six tri-state buffers. The alternative (3 pass transistors) has an 'on'-resistance that is too high to be practical [90].

We illustrate the effect of EESC in Fig. 1.6. With wire sectioning, when the 3 sections between *t0* and *t1* are driven during a particular cycle, the sections leading off to *t2*, *t3* and *t4* are not driven. This saves the energy required to change their voltage. If additionally the short

sections between *t0* and *t1* are at once the shortest and the most active ones over the duration of the program, the energy-saving effect is stronger. We call this *power-aware placement*.

The concept of EESC involves at once a style of interconnection, that can be compared with other styles like network-on-chips (NoCs) and shared media, a design framework, including macro-based layout and activity-aware floor-planning, and a preference for programmed control over hardware control. Besides consuming little energy, on-chip communication must scale well with the number of channels and terminals involved in communication, display low latency and little wire congestion, and suit automatic design procedures, like floor-planning, well. For all these criteria, the EESC concept, as it turns out, scores very reasonably. Nowhere is it perfect. EESC is intended to be the best compromise for intra-tile communication systems over the medium-distance in the DSM domain.

Accounts of EESC as a methodology for power conservation have been made by IMEC's Technology-aware Design (TAD) group in various publications [46, 47, 48, 49, 91, 116] and two PhD theses [50, 92]. High savings were achieved in energy conservation through resource partitioning, block ordering and energy-aware floor planning. On average, EESC can decrease communication energy consumption by a factor of 2.9, when compared to a shared-media bus approach without sectioning [50]. This impressive performance can be seen as the result of applying heterogeneity with high granularity, based on decisions made at a high level (programmed control), with little overhead, on an architecture that was partitioned well.

1.1.4 The Problem of Control

Previous work by TAD studied interconnects in the *data plane*.[2] In [92], it was recognized that controlling EESC poses scalability problems, but these were not yet addressed. This thesis, focusing on the *control plane*, will fill in what was hitherto a gap in our mastership over EESC, addressing its problems of control, and confirming its feasibility, which

[2] In telecommunications, functions associated with providing services in networks are commonly split in *data*, *control* and *management* planes. The data plane is associated with end-to-end transport of data. Functions of control and management plane, in the conventional sense, refer to distributed and centralized control information respectively. We follow the less rigid usage of not making this last distinction and calling both "control".

depends much on the method of control: the switches in Fig. 1.6 have to be driven by the proper control signals.

When this work started, it was not known *(i)* from where the control bits for the sectioning switches should originate: fetched by value from memory, or generated from the program's control flow in some way; *(ii)* which cost elements dominate the control process; *(iii)* whether control information can be processed with advantage for distribution over the surface of the processor; *(iv)* whether temporal re-use can be used profitably for control of EESC; and *(v)* whether it is possible in all circumstances to generate correct control information. Also, the energy requirements for control can conceivably undo the gain from sectioning. This must be avoided. Finally, even if the energy balance of costs versus savings is favorable for small and simple networks, the method of control might scale badly to very large or complex networks. Thus, the main challenge is to establish a framework achieving control at low cost, for a range of communication architectures that might ultimately become large and complex. This particular aspect had not yet been researched at IMEC, and became the subject of this thesis.

1.2 Contributions and Publications

In this section, we highlight the various contributions made to this body of work by individual persons, at IMEC and elsewhere, including the author.

Origins Bus segmentation[3] has a long lineage [11, 5, 77]. It has been applied for resource conservation at every scale of computing: for inter-rack connections in supercomputing, where the concept originated, to inter-board, on-board and on-chip level. Presently we use it for wires less than a millimeter long.

EESC was conceived at IMEC [89] by derivation from earlier work on segmented buses, e.g. J. Y. Chen's [27]. What makes control for EESC demanding is not the mere fact of sectioning but its high granularity, in space and in time. Said otherwise, what distinguishes EESC from other forms of bus segmentation are the short length and the high number of

[3] In literature, bus segmentation is also called "bus isolation" or "bus splitting". In this thesis, we want to avoid using the term "bus", which has become ill-defined, so we use henceforward the term *sectioned communication*.

sections, and the frequent reconfiguration.

H. Zhang [124] published a seminal paper on on-chip communication, applied to digital signal processor (DSP) chips. He proposed some sectioned interconnect architectures suitable for exploitation of heterogeneity and parallelism, both for local and global connections. Zhang's paper discusses hierarchical networks, multiple buses, and elements of power-aware placement. Many other papers [6, 27, 66, 70, 79] on on-chip segmented buses address energy-related issues, and advocate the savings that can be obtained from sectioned hierarchical topologies. The topic of how to control these structures is not explored yet. For control, their authors often rely on various forms of arbitration. Contrary to this view we think that, in the context of intra-tile SoC communication, such an approach involves unnecessary overheads, and will elaborate on this view in Chapter 3.

1.2.1 Contributions at IMEC

Since the evolution of our thinking makes for an interesting story, especially for a future implementer, we mention the contributions in historical fashion. The principle of EESC emanates from IMEC's unified design flow for low-power data-dominated multi-media and telecom applications [24]. It was developed by a TAD team including Miguel Miranda, Antonis Papanikolaou, Hua Wang and Jin Guo, who performed the work in the data plane, including power-aware placement. Antonis Papanikolaou, my daily supervisor at IMEC, and myself decided on the first strategies for exploration of control and the methodology for experimentation.

Statistical studies on application drivers yielded an insight in bit activities in the control plane, and on the prospect of clustering the control plane by proximity of switches. Florian Starzer did power-aware floorplanning for these applications, using the Data Access and Storage Management for Embedded Programmable Processors (DTSE) methodology [25]. Scheduling and profiling were performed with IMEC's ATOMIUM Low Power Design Tool, while I myself developed a program named Segmented Bus Analysis (SBA) for profile analysis. Results, published in [56, 57], showed *(i)* that transport energies of control were reasonably low for small fixed-bandwidth communication architectures, and *(ii)* that clustering the control plane by proximity of the switches was much less required than clustering by bus. Some

disadvantages of the method of analysis showed up. With SBA, a margin of error exists in attributing cycles in the communication system to basic blocks in a program. Also, overlong execution times are required for profiling and analysis.

Finding it difficult to separate run-time concerns from compile- and design-time considerations, and to distinguish between the roles of designer, compiler, instruction set, program, and processor at different points in time, we developed the paradigm of an idealized "communication-aware" processor. This paradigm is novel even if maybe not revolutionary. It helped us to settle questions, doubts and arguments. Using this paradigm, we made a detailed study of a single-issue microprocessor with a memory hierarchy and loop buffers, capable of running simple applications. This platform was an archetypal communication-aware processor. Physical models for the wiring were from Banerjee[13]. The memories were modeled with CACTI, an existing model for memory access time, area, and power [22]. Hao Zhang did the power-aware floorplanning. Grunwald [43] provided a transaction-level model of a DLX processor. I wrote the register-transfer level (RTL)-level model of EESC and the System C simulator. As a result, we had a detailed, full and working representation of all aspects of EESC, including compiler and processor functionality. This ensured that we were not overlooking any cost element. The work was published in [59] and the method of simulation presented in a lecture [61]. Gradually, I developed new ideas on the relative importance of loop buffers and instruction set architecture, on the scaling behavior for large and complex topologies, the opportunities of topological analysis and on useful-state analysis. This involved a paradigm shift from software-assisted to program-controlled operation, and enabled an optimal control plane to emerge. The need for a special-purpose compiler disappeared, and attention shifted away from shared-media "buses" towards irregular topologies.

Loop buffers Initially, it was thought that the control information for the control plane would be read by value from instruction memory. In that case, they would form part of a 'communication instruction', to be fetched for each transfer. Loop buffers, which play a role in low-power instruction memory organization (as described in [68]), were thought to have a specific role in the EESC control plane.

Gradually I realized that loop buffers must be employed with one of

two purposes in mind: to reduce instruction fetch energies (I called this "fetch loop buffering") or to reduce power dissipation from transporting control information (which I called "transport loop buffering"). It was found that the two purposes cannot be combined in a single loop buffer, since the decoding circuits of the processor get in the way. If both purposes were to be implemented, one would need two sets of loop buffers.

Fetch loop buffering is orthogonal to the issue of EESC. As for transport loop buffering, it can only reduce vestigial control transport energy in the processor, after fetch loop buffering has done its work. This was found unnecessary, since experiments showed that: *(i)* per control wire, control transport energy is not higher than data transport energy. As long as the number of control wires is small, control transport energy will not dominate. *(ii)* Memory access for transport loop buffering turns out to have an unacceptable energy cost overhead; and *(iii)* generating control bits with a path decoder is in comparison almost free. The concept of control processing with loop buffers, which had been central to our thinking, lost its primacy.

Instruction Set Architecture At first, it was assumed that "software control" by itself meant a responsibility for the compiler to calculate control bits. Since no compiler designers were part of our team, this posed a problem. Gradually we realized that the compiler does not always have special tasks in connection with control of EESC: what we will call "transfer-awareness" is already part of the instruction set architecture (ISA). Scheduling is already done by a compiler. Communication- or topology-awareness (these concepts will be defined later) have their merits, but only if they are already part of the ISA. Hence, the ISA is the medium of communication between the designer of the control plane and the writer of the compiler. We could thus still do programmed control in the absence of any special-purpose compiler, if only for fixed-bandwidth communication architectures. For variable-bandwidth scheduling, a theory (useful-state analysis, described in Chapter 4) had to be developed before any compiler could be built.

Large and complex topologies Shifting our attention to more complex communication architectures, we became aware that the problem of scalability was not necessarily linked to topological complexity but

rather to size of the network, in terms of number of switches. The transport energy in the control plane was not by itself so much of a problem as was the bitwidth of the control plane: this width represents a cost in chip area, for simple as well as for complex topologies.

Topological analysis Our early methodology centered on lengths of wire sections and transition activities on the wires of the communication system. These are cumbersome to analyze because they result from extensive interaction between research concerns (optimization of the communication network), design activities (chip layout), compilation (ISA and control decisions to be made), run-time activity (physical phenomena) and post run-time analysis (profiling, which then couples back to design).

It was realized that many aspects of the data plane could be decoupled from control plane design. A baseline gain could be aimed for by design of the topology, making abstraction of geometry, activities and schedules. In this approach, the topology is seen as determined from data-plane design work. The control plane then follows a design framework described in this thesis, which is optimal in the absence of data about geometry, activities or schedules. It guarantees a certain gain from EESC, that can be improved upon by better statistical analysis, but is in the first place an achievement of topological analysis itself.

Useful-state Analysis Regarding the scaling with large topologies, we realized that we could reduce control plane width by a minimal-redundancy representation of the communication network to be controlled. This would lead to a reduction of both control fetch energy and transport energy. Wire congestion would also be reduced. We developed a theory of network control that had the advantage of being general. Finding this theory (of useful-state encoding) was surprising since it is powerful in reducing the size of control circuitry and yet not to our knowledge described elsewhere in literature. It his original work, arrived at by myself with hints and suggestions of my promotors; Grover [41] and Panconesi [88] provided inspiration for the algorithms, further developed by Peter Veelaert (who inspired the FAR algorithm) and me.

Having separated statistical and topological analysis, and developed Useful-state Analysis (USA), we applied these in a second optimization to the control plane that we already had developed for [59].

We could now also analyze the scalability of more complex systems, avoiding time-consuming placement and profiling and resulting in a publication [62].

The design pattern emerging ultimately differed substantially from our early ideas. USA emerged late in the day, thus we did not do the combined statistical/topological analyses that it enables. Even so, the limits of scalability of EESC have been pushed up appreciably during this work, as the use cases in Chapter 8 will show.

Design support software Analyzing larger communication architectures led us to realize the limitations of early rapid-prototyping software and develop a second generation of design support tools, with better performance and applicability over a larger range of problems.

1.2.2 Publications

The work led to the following publications:

1. [56] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Energy costs of transporting switch control bits for a segmented bus", Proc. 16th Annual Wsh. on Circuits, Systems and Signal Processing (ProRisc 2005), pp 359-364, Nov 2005.

2. [57] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, K. Debosschere, W. Philips, "Energy consumption for transport of control information on a segmented software-controlled communication architecture", 2nd Intl. Workshop on Applied Reconfigurable Computing (ARC 2006), LNCS 3985, p. 52-58, Mar 2006.

3. [58] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Network control, topology and transfer scheduling for synchronous system-on-chip communication", Architecture and Compilers for Embedded Systems (ACES) 2006, Ghent University, pp. 42-45, Oct 2006.

4. [59] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Using a linear sectioned bus and a communication processor to reduce energy costs in synchronous on-chip communication", Intl. Symp. on System-on-Chip (SOC 2007), p. 117-120, Jan 2008.

5. [60] K. Heyrman, "Using SystemC for the power simulator of an on-chip sectioned bus", http://www-ti.informatik.uni-tue\
-bingen.de/~systemc/Documents/Presentation-SM07-UP2_
heyrman.pdf, European System C Users Group Special Meeting (ESCUG-SM), Nov 2007, Tampere, Finland.

6. [61] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Control of low-power synchronous on-chip communication", Advanced Computer Architecture and Compilers for Embedded Systems (ACACES 2007), pp 29-32, Jul 2007.

7. [62] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Power gating for wires", IEEE Trans. on VLSI Systems, (accepted for publication).

1.3 Overview

In this introductory chapter, we have discussed the problems of increasing energy consumption caused in interconnects by technology scaling, specifically for embedded SoCs.

The remainder of this work is organized as follows: Chapter 2 studies the landscape, of on-chip communication, focusing on control, and the niche that EESC occupies.

Chapter 3 lays out strategies for optimization of control: it provides a physical model for wire sectioning, exposes the consequences of programmed control, surveys the nature of communicating entities on chips, and provides the principles that guide optimization. It also discusses architectures where our concept of control over EESC fits in, or where similar concepts are advocated.

The core of our work is contained in Chapters 4- 6. Chapters 4 formulates a theory of useful-state encoding (USE) to obtain minimal-size encoding of control information, and Chapter 5 contains the algorithms developed for control of EESC, and their implementation. Chapter 6, which applies USE, following the principle of programmed control, to a design framework for intra-tile communication, wherein the cost factors of control are identified.

Chapter 7 discusses past and future methods of using our framework for analysis, simulation and design. Chapter 8 contains use cases of these methods, for different types of communication architecture,

and Chapter 9 presents the conclusions of our doctorate work and suggests future avenues of research.

For the benefit of users of our design framework, Appendix A describes design support tools for the control plane while Appendix B contains circuit diagrams and sample HDL code for components of the control plane. A concise index, mainly intended for retrieval of original definitions, is provided at the end of this work,

Chapter 2

On-Chip Communication

I'faith, there's time for nought but bold resolves!

John Barth, *The Sot-Weed Factor*.

Having situated the purpose of our research, we now take a closer look at on-chip communication, concentrating on features relevant to control.

Then we will study some alternative solutions, and interpret comparisons that have been made between rival approaches, From this comparison, we obtain objectives for control over EESC.

BASED on literature, the following overview discusses existing philosophies for control of on-chip communication. In order to properly relate different sorts of on-chip network to our type of intra-tile communication, we will review the established taxonomy of interconnection engineering. We then discuss individual control methods relevant to on-chip communication. These include some we will not apply, like arbitrated buses and NoCs but that must be contrasted against our solution. We will then present some comparisons between rival design solutions, exposing fundamental differences in control methodology. This will lead in the next chapter to our own principle: programmed control of EESC.

2.1 Taxonomy

In our study, we survey networks where the granularity of control is similar to ours. A tendency exists, in the nomenclature of communication systems for aspects of implementation to denominate the whole. This can lead to other aspects being overlooked. We ourselves want to discriminate, in each publication, what the implications for control are. The following classification of communication networks shows that networks can differ in many ways. It was proposed in [83], and reworked in [31] and [69]. Requirements for control are influenced by all aspects mentioned below.

- On-chip networks are *not* single-commodity transport networks.[1] Individual transfers between terminals interact, making our control problem a *multi-commodity* flow problem [2].

- All networks are *reconfigurable*, or they would be just a set of dedicated connections. We call a network *frequently reconfigurable* if the volume of reconfiguration control information is not negligible w.r.t. the volume of transported information. Since the high volume of control information is expensive, the fine grain of reconfigurability with EESC causes us to make this distinction.

- In interconnection engineering, networks have traditionally been classified according to *operating mode* (synchronous or asynchronous)[2] and to *localization of control* (centralized, decentralized or distributed, i.e. with multiple centers of control). Since a tile is controlled by a single program, intra-tile architectures normally implement synchronous communication.[3] Programmed control is conceptually centralized, even if it is physically distributed over the tile. Many architectures of interest today (like the Internet, multicomputer, multiprocessor and some chip-level

[1] Some networks transport a single commodity, like oil or water, from sources to sinks, in some optimal fashion. In contrast, communication networks treat transfers between each pair of terminals as a distinct commodity, making the networks multi-commodity.

[2] The distinction between synchronous and asynchronous refers to events occurring at both ends of the links. If they are not locked to the same clock, some form of handshaking is required, and the operating mode is asynchronous.

[3] Asynchronous and self-timed stored-program processors are feasible [86] but not part of the SoC mainstream.

multiprocessor (CMP) networks), implement *asynchronous networking with distributed control* [29, 31]. The main distinction, then, between our context and the main body of contemporary interconnection engineering is that our control can (and will) be *synchronous and centralized*.

- The distinction between *circuit-oriented* and *buffered* networks is also very basic. In circuit-oriented networks, a direct circuit completes a path between inputs and outputs. As a result, data can be transferred directly from input to output at high rate, depending solely on the characteristics of the path in-between. In packet-oriented networks, buffers are added to the circuits. The mode of operation is "store and forward". Data can be memorized at the network vertices for later transmission. This decouples the allocation of adjacent links, possibly allowing more efficient flow control. However, control now needs to allocate buffers as well as link capacity. In on-chip communication, buffered networks are called NoCs.[4] EESC will be circuit-oriented.

- Many classifications have been performed on the basis of the underlying *topology*. Some topologies are quite general (linear, grid[5], torus, ring, hypercube, tree), some are specialized (De Bruijn network, star graph, K-ring, butterfly, octagon...). Some are *regular* (they can be decomposed into orthogonal dimensions, like grids, tori and hypercubes); some cannot, being *irregular*. Since the topic is extensive and the choice between different topologies really a data-plane issue [87], we have in our work successfully decoupled issues of control from issues of topology. Consequently, we can handle any topology class with our concept of control.

- Distinction is often made between three *modes of resource sharing*: *shared-media networks*, *direct networks* and *indirect networks*. Shared-media networks share all (or some critical) resources of a communication resource set. Direct networks (also called "point-to-point") use one resource per link. Indirect networks interpose several intermediate resources between source and destination. *Hybrid direct/indirect* networks have also been described. These

[4]The unwritten assumption is that NoCs are always buffered and packet-oriented. This is part of industry usage.

[5]Many authors use the term *mesh* for our *grid*. Following the example of [41], we will reserve the term *mesh network* for any network that contains closed paths.

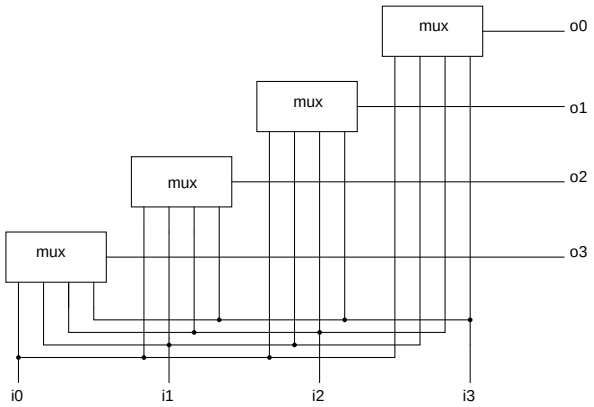


Figure 2.1: A logical direct ("point-to-point") network may physically be direct (multiplexers).

comprise *hierarchical* and *multiple* networks. The criterion of resource sharing depends much on what the resource is. It could be as small as a transistor switch, or as complex as a router.

To qualify the distinction between resource sharing modes, the following observation must be made about the layer of resources where it is applied. In multi-layer networking, a difference is often observed between the high-level connection topology, called the *logical network*, and the actual physical connections, the material layout of the net, making up the *physical network*. A logically direct network (point-to-point), can be physically direct or indirect. For instance, a logical "point-to-point" can be implemented by a set of multiplexers (Fig. 2.1), which is physically direct, but also as a physical crossbar which is physically indirect. (Fig. 2.2). Control is of course different in both cases. In practice, the terms of "point-to-point", "network of multiplexers" and "crossbar" are often used interchangeably. We will do the same, using the quote marks to indicate the relativity of the distinction. In our theory, in Chapter 4, we will treat the distinction between logical and physical connections by discriminating between the logical *set of transfers* between terminals, and the physical *network topology* between terminals and resources.

- In older literature [20], distinction is often made between *single-* and *multi-stage* networks. If a "stage" is seen as a direct or "crossbar" network, then multi-stage networks consist of a cascade of

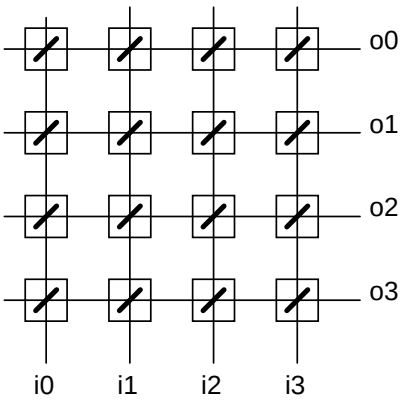


Figure 2.2: The same logically direct ("point-to-point") network may also physically be indirect ("crossbar").

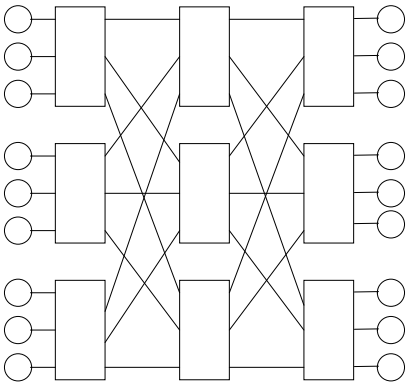


Figure 2.3: A multi-stage topology consists of a cascade of "crossbars".

crossbars (Fig. 2.3). The purpose of multi-stage networks is to improve scalability over that of full crossbars, at the expense of latency. Since we make abstraction of topology, the distinction is not essential to us.

- Variations exist in the ability of networks to establish new connections, and various terms have evolved to describe them: *non-blocking* (in the *strict sense* or in the *wide sense*), *rearrangeable*, *blocking*. (For a good classification, see [20].) The latter, for instance, means that connection sets exist that will prevent some additional connections from being established between unused ports, even with rearrangement of the existing connections. The reason for

the variety in terminology is that the criterion not only involves network topology, but also use and scheduling policy of communication resources. The property of being blocking or non-blocking therefore belongs to a communication architecture as a whole, and not to the network topology alone. We will call the property of a communication architecture that corresponds to the "blocking" or "non-blocking" criterion, which is difficult to define for the physical network, the property of a communication architecture of respectively having "variable" and "fixed" bandwidth. This is an important property for communication architectures, but its rigorous definition will have to wait for the mathematical definition of a communication architecture, given in Chapter 4 (Section 4.4). The reason why we need a new definition is that with centralized control, the scheduling entity (a compiler, in our case) schedules transfers *collectively*. This is not the common case: most networks found in literature are scheduled under distributed control.

- With centralized control, the issues of *deadlock* and *starvation*, prevalent in many network classifications, are not our concern. With collective scheduling, not avoiding them is simply a design error.

- What does concern us, if only for the purpose of comparison, are issues of *resource contention*. *Arbitration*, which resolves resource contention, can be based on *priority*, *tokens*, or other mechanisms and protocols. With centralized control, the issue does not arise any more after scheduling.

- The control plane is influenced by the type of *payload* that the network carries. Payload may be of *message* type, where each item is self-contained and unrelated to other messages, or of *stream* type, where each is part of a sequence that must be delivered in order. This distinction influences optimal arbitration and routing. EESC, operating within the tiles of SoCs, is concerned mainly with message payloads.

- In packet-oriented architectures, the *size* of the payload may be *fixed* or *variable*. *Packets* are atomic units in the sense that sufficient buffering must be provided to either transfer a packet in its entirety or delay transmission until buffer space becomes available. Packets can be subdivided in *logical* and *physical message flow*

control units[6] (*flits* and *phits*). In contrast, inside a stored-program processor, the size of the payload, if variable, is still fixed in increments of *subword* (usually byte) size, and limited to *word* size.

- The payload can also be of *addressed* type, when each item is associated with an address that implicitly positions the data in a space (for instance, a memory space or a register file). This has consequences for control (for instance, addressing may play a part in terminal selection logic) that will be explored in Section 6.1.

- *Collective communication* schemes have been identified, sometimes classified in terms of multiple one-to-one, one-to-all, all-to-one and all-to-all communication, or of broadcast, scatter/gather, collection and dissemination. In intra-tile communication, simple forms of collective communication will need to be accommodated. We will use the terms *broadcasting* for all-to-all and *narrowcasting* for all-to-some communication.

We have chosen not to mention some other common categorizations, most of which pertain to packet-oriented networks. By now, the reader will be aware that comparisons between communication systems must be made prudently, especially with regards to operating mode (synchronous or asynchronous), localization of control and circuit- or packet-orientation.

2.2 Methods of Control

Despite our intention to use programmed control, which is unlike the control over HDL-generated multiplexer networks, "crossbars", arbitrated shared systems and NoCs, we must briefly discuss how those are controlled, if only to mark the differences with our work.

Synthesized communication logic In circuits generated from Hardware Description Language (HDL) descriptions, signal wires are often declared as persistent and dedicated communication resources. Since such resources are not reconfigurable, the design does not yield a "network" in our sense. In HDL-based designs, alternatively, existence

[6] A flow control unit refers to that portion of a packet whose transfer can be synchronized.

of multiplexers can be inferred from circuit descriptions using native grammar. Multiplexers can also be introduced by HDL compilers in the mapping phase of synthesis. For instance, some logical wires can be multiplexed into single physical wires using a technique termed *virtual wiring* [10]. Multiplexers, then, are the network resource of choice for synthesized circuits that are manually coded. In comparison to EESC, control over multiplexers is relatively simple, and amenable to programmed control as well. But for efficient resource usage, multiplexers are inferior and not scalable.

Since ordinary multiplexers[7] are not optimal for low-power, specialized Intellectual Property (IP) blocks are often used to achieve low-power on-chip communication. Four types of IP can be distinguished: *(i)* very large-scale integration (VLSI) versions of "crossbars", *(ii)* single or multiple shared-media architectures, commonly called "buses", *(iii)* NoC architectures, and *(iv)* our own solution, RTL-level components for programmed control of EESC. The latter have been defined as parts of our simulator, and are described in Section B.1. In this section we concentrate on alternatives to EESC.

VLSI "crossbars" [32] describes versions of the classic multiplexer-based "crossbar" (cfr. Fig. 2.1) that take advantage of optimizations and design choices possible in VLSI. The problems of the scalability of large "crossbars" have often been highlighted [31]. Y. Zhang [123], in an influential paper, compares interconnection fabrics between functional units, of two types: point-to-point ("crossbar") and multiple shared-media ("bus"). His model is suited for intra-tile communication: the network is synchronous, and the author forgoes arbitration. Calculating power and delay (both for perfectly scheduled and for randomly generated data sets), the author concludes that "crossbars" have an advantage when the number of terminals is small and good scheduling has been achieved. A penalty in area always exists: the basic scaling law for size in function of the number of terminals N is $O(N^2)$, and can only be improved (e.g., to $O(N \log N)$, see [31]) by increasing latency. Large VLSI crossbars, then, are not an option for intra-tile SoC communication, although controlling them is simple.

[7] Multiplexers with output disable *can* be used for sectioned communication, see Section 4.1. Scalability problems remain.

Arbitrated "buses[8]" Many designers of low-power on-chip communication will turn to a set of related IP that provides an inventory of bus components. Some of these sets are commercial: CoreConnect from IBM, Advanced Microcontroller Bus Architecture (AMBA) from Advanced RISC Machines (ARM), or STBus from ST Microelectronics (STM). Some originate from industry consortia (OCP-IP from OCP International Partnership), or are open-source (Wishbone from `opencores.org`). All of them are conceptually committed to arbitrated control.

An example design featuring an arbitrated-bus communication system, based on commercial IP fabric, is found in [71]. The design uses the AMBA IP set which provides three different kinds of bus architecture for different applications: the peripheral bus (APM), system bus (ASM) and high-speed bus (AHM). None of these sets are geared specifically to sectioned communication. Some form of coarse-grain sectioning can be achieved by combining such buses. In [71], Advanced High-performance Bus (AHB), the fastest sub-architecture, was used for intra-processor communication. A system featuring AHB typically consists of master modules (e.g. processors), slave modules (e.g. memory and communication modules), and an AMBA interconnection core.

The *standard* AHB implementation provides fast communication between several master and slave modules; it is capable of pipelined operations, burst transfers and split transactions. The physical connection between master and slave is realized by multiplexers. A central arbiter grants the masters access to the bus and a decoder selects the active slave. Only one master is allowed to access slaves at any time, thus the standard AHB is ill-suited for multiple instruction-issue processors.

The switched version of AHB, named *multi-layer* AHB by ARM, is a different realization of the architecture and allows multiple concurrent transfers between several pairs of masters and slaves. The interconnection core for the multi-layer AHB is very different from standard AHB: instead of one decoder, there has to be one decoder for each bus master. The central arbiter of standard AHB is replaced by an arbiter for each slave, preventing more than one master from accessing the same slave at the same time. A module can be master, slave or master and slave at the same time. In the last role, the module is capable of performing

[8]Many authors consider it an defining property of a bus to connect masters and slaves, and to feature arbitration units and bus protocols. For the remainder of this chapter, we will join them in this usage.

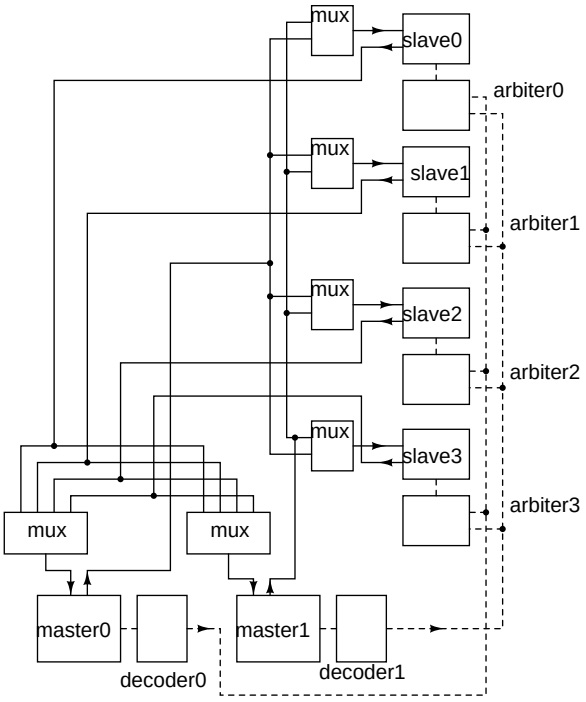


Figure 2.4: Multi-layer AHB "crossbar" with 2 masters and 4 slaves. (Source: [71]

master accesses like DMA, beside being accessed as a slave.

From a block diagram of a multi-layer AHB-based system, shown in Fig. 2.4, we can judge the inherent overhead of the arbitration system. Arbiters are needed for every slave, and encoders for every master. The need for synchronization of arbitration slows down multi-layer AHB in comparison to standard AHB: e.g. for random-access read transfers, the transfer rate is effectively halved [76]. In contrast to EESC, this need for arbitration will prove a disadvantage.

Networks-on-chips From 2001 onwards, a new paradigm for SoC communication was developed [16, 19, 28]. The motivation for NoCs was presented in [28] as follows: SoC communication takes place over the surface of chips with diminishing feature sizes. With this scaling, ever more terminals communicate, over distances that become comparable in size (with wire sizes and repeaters taken into account) to the wavelength of clock frequencies. Existing network design techniques

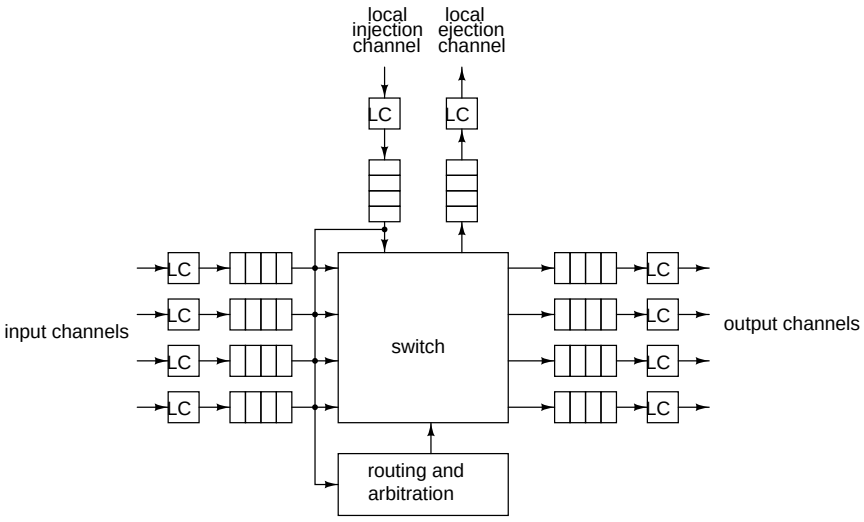


Figure 2.5: Generic router model showing buffers and link controllers (LC) Source: [31].

scale badly with the number of terminals, but buffered networks have inherent advantages in run-time scheduling over circuit-oriented networks. The authors of [28] thus advocate to design general-purpose long-distance on-chip networks, well-engineered, with controlled electrical parameters, so that low power and high-speed operation is guaranteed. These optimal wiring resources can then be shared between many functions.

The authors of [16, 28] stress that the overhead in area for a terminal with router box is limited to the order of 6-10% of the area of a terminal. The fact that links are segmented between routers amounts to a form of wire sectioning. Fig. 2.5 shows a generic model of a NoC router box. Routers contain first-in first-out (FIFO) buffers and a link controller per in- and output, and a central switch controlled by a routing-and-arbitration unit. If an output link is requested by an input link controller but is busy, the incoming message remains in the input buffer, to be routed again after the output link is freed and the input link successfully arbitrates for the output link. NoCs use *circuit-* or *packet-switching*[9], to minimize the latency caused by store-and forward buffer-

[9]or other types of switching, such as *virtual cut-through*, *wormhole* or *mad-postman* [31]. Circuit- and packet-switching are not each others opposite: distinguishing between the two does not distinguish between NoCs and EESC. Rather, both

ing. The networks route packets over *virtual channels*. Virtual-channel flow control improves latency and throughput by allowing messages to share a physical link. Tools for stochastic study of communication patterns have been imported from distributed Internet and multiprocessor networks to quantify the scalability of NoCs. Multilayer protocols, fashioned after the ISO 7-level model, allow communication services with different quality of service (QoS) to be defined, or reliable communication to be implemented even in the presence of datagram dropping and misrouting. The services can discriminate for instance between payloads, according to stream or message or other type.

Resources are allocated by run-time arbitration, unless they can be pre-scheduled. Although NoCs have certainly been used for intra-tile networks, we can see in the scheduling process an inversion of normality. Intra-tile, most traffic is pre-determined by the program. Some I/O and memory operations are handled by exception routines (cache and page misses, buffer under/overflows, interrupts). In NoCs, it is exactly some of this exceptional traffic that can be pre-scheduled (i.e. streams) while the normal traffic (memory references, i.e. messages), is scheduled 'dynamically', i.e. as an exception.

NoC routing To contrast routing in NoCs to our own methodology, it is instructive to sketch the way that routing decisions like the one shown in Fig. 2.6 are made. The figure shows two paths between two pairs of terminals. The paths shown happen not to share nodes (router resources) but, since the nodes of Fig. 2.6 are store-and-forward, they could well do so, as long as the nodes are not shared *in the same cycle of operation*. The paths are realized over multiple cycles, with buffering in-between. Paths can be established by a controller at the source node prior to packet injection (*source routing*) or determined in a distributed manner while the packet travels across the network (*distributed routing*). Source routing requires the inclusion of complete route information in the packet header. Distributed routing can have a more compact header, since only the destination must be encoded in the header.

Routing algorithms abound [29, 31]; they are commonly implemented by consulting a routing table (*table lookup*) or by executing a routing algorithm. If the topology is regular, the routing algorithm can be run in hardware, using a finite-state machine. Networks with

circuit- and packet-switching, in addition to all other types of switching, are fundamental design choices for NoCs.

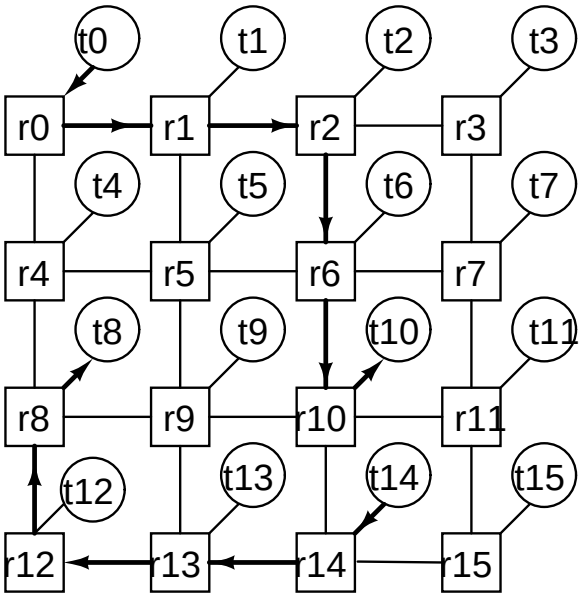


Figure 2.6: NoC routing example on a two-dimensional grid.

irregular topologies, using either source or distributed routing, must have a lookup table at each node. Table sizes increase (in principle) with network size. Routing algorithms may supply a single path for each transfer (*deterministic routing*) or, alternatively, consider network state while making a decision (*adaptive routing*). The latter is in practice compatible only with distributed routing.

From this (very brief) review, it comes across that scheduling and routing in NoC is essentially a run-time process, so that its costs are continuously recurrent. Our form of programmed control will incur these costs at compile time, as much as possible.

2.3 Comparisons of SoC Communication

Comparisons of different sorts of SoC interconnection systems have been the subject of a number of papers [4, 76, 81, 102, 103, 123]. This section presents four of those, exposing strengths and weaknesses of different control methodologies. Such a comparison can be difficult to interpret, partly because some aspects of communication control cut straight across the space of exploration. To avoid silent assumptions

and unwarranted conclusions, we will use the categories from Section 2.1.

Lahtinen [76] makes a comparison of **two interconnection architectures with arbitration**, and studies the way area, bandwidth and latency scale with the number of terminals. His two implementations are: *(i)* a single shared-media network with multiplexers, and *(ii)* a physically indirect, logically direct network with 2 "buses", built from 12W4T (12-way, 4-terminal) switches, not dissimilar to the sort of networks we consider in our work. Both are built from STM IP. The network is intended for synchronous communication. The arbitration is accomplished with a simple non-interruptive priority algorithm [45], where ownership is given to the master with the highest priority, but if an agent starts a transfer, it cannot be interrupted by a higher-priority master. According to Lahtinen, arbitration is the most complex process in either implementation and can take up a lot of time. It is not desirable to let the arbitration limit the global clock cycle, and thus it must be allowed to take several clock cycles. The arbitration is constant-time for *(i)* but for *(ii)* scales badly with more terminals, because of the set-up time of the switches and the need to keep track (with this arbitration protocol) of ongoing transmissions .

Mäkelä [81] compares **four interconnection IP fabrics between functional units**, in the context of Transport-triggered Architectures (TTAs). TTAs are synchronous, intra-tile, and scheduled at compile time [64]. The four fabrics are *(i)* a medium shared by tri-state buffers, *(ii)* a medium shared by means of an AND/OR structure, *(iii)* a medium shared by means of a multiplexer, *(iv)* a indirect network called by the author a "segmented multiplexer bus." We will call it a "distributed multiplexer architecture," as done in the Wishbone IP set [120], where such a structure is also used. The delay and power consumption of the fabrics, in function of the number of functional units (terminals), are compared. In his conclusion, Mäkelä indicates that fabric *(ii)* seems to offer the best alternative in scalability of delay and power consumption, while *(iv)* has the highest delay (because of cascaded multiplexers), but that its strength may lie in the possibility of realizing multiple transfers at once, on a single "bus". We, for our own purposes, note that *(i)* TTAs use, or indeed need, no form of arbitration; *(ii)* none of the fabrics offers sectioned communication in our sense, but that all of them could be modified to do so; *(iii)* scheduling in TTAs is not topology-aware (see also below, Section 3.2.4).

Angiolini [4] compares **three asynchronous multi-processor interconnection architectures** (not intra-tile), each with 30 IP cores (10 masters, 5 traffic generators, 15 slaves). The three configurations are *(i)* a single-layer ("shared-bus") AMBA AHB configuration *(ii)* a multi-layer AHB with 5 "layers", i.e. shared-media subnetworks, and *(iii)* an *xpipes* [108] NoC with a 3x5 grid topology. Two stream-oriented benchmarks were considered. The first one is highly parallel, with cores operating independently of each other. The second one performs heavy inter-core synchronization, since it modeled a producer/consumer pipeline. In the results, alternative *(i)* is found to be clearly unsuitable because of poor performance. No decisive advantage is found for either alternative *(ii)* or *(iii)*.

The NoC concept is found by Angiolini to have the advantages of virtually unlimited scalability and the ability to be deployed in arbitrary topologies. Its downside is the overhead due to packetization affecting area, power and latency. Indeed, in the view of Angiolini, the real metric to assess the speed of an interconnect is the latency from request to completion of an transaction.

With the cores in the system running at a 400 MHz clock frequency (and the NoC at maximum frequency of 885 MHz), both architectures *(ii)* and *(iii)* show the weakness of high latency, varying between 5 and 10 CPU cycles. The latency is different for single reads/writes and stream transfers.

The AMBA protocol is especially weak for consecutive single writes, since a master must wait for the previous write request to finish and then apply for rearbitration, before issuing the new write. The NoC can take advantage of stream transfers, where the packetizing overhead is only paid once and congestion becomes the key limiter. Angiolini strongly suggests that NoCs can and should take advantage of stream transfers. We ourselves conclude that this level of latency is due to arbitration in both cases, and inadmissible in intra-tile networks.

Ryu [102] makes a detailed comparison of **five multiprocessor SoC interconnection architectures**, each employing 4 PowerPC cores and a scratchpad hierarchy of SRAM memories, based on industry IP, all of them using run-time arbitration. The network is asynchronous. The throughputs for two data-intensive pipelined application programs (OFDM and MPEG2) are compared. We will not go into the specific configurations of each architecture, but would like to stress the fact that different designs, when optimized for the specifics of each IP set, force

very different topologies and memory partitioning on the architecture. Even so, differences in throughput are slight for one application (OFDM) and marked for another (MPEG2). Performance differences between arbitration protocols can be observed but not attributed to specific properties of application programs. We conclude that the performance of arbitration in industrial IP interconnection fabrics is difficult to quantify, and that optimal topology is tightly linked to processor features.

2.4 Conclusions

Concluding this tour of the state of the art of on-chip communication, we remember that hand-crafted HDL descriptions of on-chip communication at best yield multiplexer architectures without sectioning. At worst, they result in dedicated signal wiring. Both have problems of scalability. VLSI "crossbars" also have this problem, and can only be used for a small number of terminals. Arbitrated buses and NoCs alleviate the problems of scalability with number of terminals. Both solve the problem of resource contention by means of run-time arbitration. Methods of arbitration can vary much and are difficult to compare. Arbitration consumes considerable chip area and time. In particular, it causes latency of several CPU clock cycles, which can be avoided in program-controlled circuits. NoCs are better for stream-type than for message-type payloads, since they feature stream-oriented services that need less arbitration. In most stored-program processors message-type payloads are common while streams are the exception.

To arrive at a motivation for programmed control, we can build on these conclusions as follows. NoCs are packet-oriented and "store and forward". Buffered networks have an extra dimension in optimization. Over the long-distance, NoCs must ultimately perform better than EESC would in a similar configuration, since NoCs scale well with many terminals. In comparison, EESC cannot work over multiple-cycle domains, thus bus frequency must be lowered if EESC is to be used. But for medium distance, with a limited number of terminals, within a program-controlled tile, the circuit-orientation of EESC can perform better than a NoC, since it avoids serious overheads in area and latency.

Arbitrated buses have some inherent advantages for addressable payloads: the ease of implementing address and data pipelines, burst transfers and split transactions. These features are also feasible in other

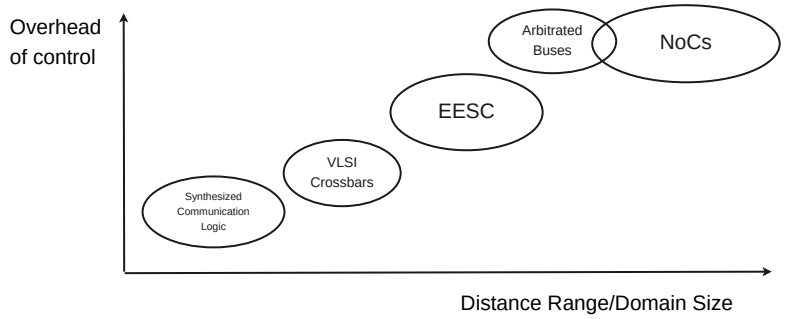


Figure 2.7: EESC occupies a niche for medium-distance on-chip communication, where control is programmed, and scalability is important.

communication architectures; in arbitrated buses, the building blocks required are part of the IP set, making for straightforward designs.

Inside a stored-program processor, we want to avoid the overhead of arbitration altogether by using scheduling information available at compile time. This avoids continually making arbitration decisions that were already made while scheduling[10]. The issue of scalability remains, so we must select network topologies that avoid the disadvantages of VLSI "crossbars" and exploit all advantages of hierarchical and parallel communication. Since a topology is normally decided upon by data plane considerations, it would be best if we could accommodate any topology.

If we succeed in this venture, we will have, in the context of intra-tile on-chip communication, an advantage over VLSI crossbars because we have more topological flexibility, and we will also have an advantage over arbitrated buses because we avoid the overhead of arbitration. NoCs, being buffer-oriented, have the ultimate advantage of an extra dimension when optimizing resource usage of the network. In intra-tile communication, they still have the disadvantage of arbitration that can be avoided, and increased latency.

Scalability of communication translates into distance over which communication can be assured. NoCs have proved their worth for distributed global communication patterns using little or no coordination. They are undoubtedly suitable for long-distance on-chip communica-

[10] Arbitration in NoCs and arbitrated buses is a distributed process. A distributed process cannot remember and reuse global decisions; there is no global context for remembering anything.

tion. Over the medium-distance, within programmed domains, with message-type data, we assert that there is a niche, pictured in Fig. 2.7, where programmed control can prove advantageous.

Chapter 3

Optimization of Control

Nous vivons des temps sans précédent, et notre obligation sacrée est de les remplir en plein.

Montesquieu.

In this chapter, we first establish a model of wire interconnection suitable to describe EESC and its optimization. EESC is seen as an extension of power gating, an established concept from low-power circuit design. We then overview the domain of intra-tile EESC, with the purpose of establishing the nature of the communicating entities, and present, in turn, a paradigm called the "communication processor", the opportunities for programmed control that it enables, the principle of optimization that it leads to, and the problems of scalability that it exposes.

At the end of the chapter, we will review processor designs from literature with architectures motivated by concerns similar to ours, and state explicitly the scope of programmed control.

ENERGY is lost in interconnection by needlessly charging sections of wire that do not carry useful information. A principle was developed by TAD to avoid this: wire sectioning, to which the author adds presently the concept of programmed control. First, we study the physical model that underlies EESC. Then we circumscribe our domain of application, intra-tile SoC communication, and the entities we find there. This leads ultimately to a paradigm and an optimization criterion to guide our design framework.

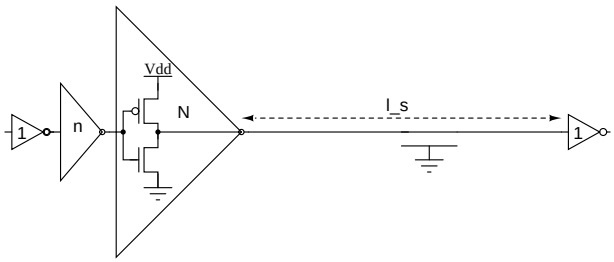


Figure 3.1: Physical model for a wire section with repeater, consisting of a 3-stage buffer.

3.1 Sectioned Communication

Wire sectioning enables us to control the capacitive loading of line drivers individually. To describe this, we adopt the following model.

3.1.1 Physical Model

In the deep submicron domain, medium and long-distance wire sections contain repeaters [13, 63, 121] consisting of 3 or 4 stages of buffers of increasing size, as shown in Fig. 3.1. The buffer stages can be dimensioned according to a tapering principle explained in Rabaey [97]. We want to model the power consumption on such wire sections. The purpose of the model is to elucidate the advantage of EESC (also known as "power gating for wires") and to provide a physical basis for optimization.

Transport energy The transport energy loss E_{xp} on a wire section is the sum of dynamic and static energy loss in the buffers. The former consists of switching and short-circuit loss, the latter is leakage. Thus transport energy loss has the following elements: $E_{xp} = E_{dyn} + E_{stat} = E_{sw} + E_{sc} + E_{leak}$. Switching loss E_{sw} and short-circuit loss E_{sc} are proportional to the activity on the section. E_{sw} is also proportional to wire capacity, thus section length, while E_{sc} is independent of this length. E_{leak} is proportional to section length but independent of activity. If s is a wire section, W the bitwidth of the section, α_s the activity factor (i.e. the average fraction of voltage flips per bit and per clock cycle, observed over the duration of a program), and l_s the wire length of the section, then the energy loss per section and per cycle is expressed as

$$E_{xp_s} = \alpha_s W l_s E'_{dyn} + \alpha_s W E"_{dyn} + W l_s P'_{stat} T_{clk}. \quad (3.1)$$

E'_{dyn}, $E"_{dyn}$ and P'_{stat} are technology factors. T_{clk} is the duration of a cycle. For a given program and input, α_s can be recorded by profiling. Accumulating the loss over the duration of a program, we find the total transport energy loss as

$$E_{xp} = N_{cycles} \sum_{\forall s} E_{xp_s}. \quad (3.2)$$

When switching loss dominates, (3.2) can at first be approximated by

$$E_{xp} = W E'_{dyn} N_{cycles} \sum_{\forall s} \alpha_s l_s. \quad (3.3)$$

Making abstraction of the activity factor α_s, it drops out of the picture. This assumption[1] simplifies the analysis even more. The value $\alpha_i \approx 0.15$ is a suitable estimate for the activity factor of every section. Such an estimate is valuable in the absence of detailed profiling, which is an elaborate procedure. Using the estimate, transport energy loss from (3.3) can be rewritten as

$$E_{xp} = W E'_{dyn} N_{cycles} \alpha_i \sum_{\forall s} l_s. \quad (3.4)$$

The part of E_{xp}, caused by leakage, is

$$E_{xp_{leak}} = W P'_{stat} T_{exec} \sum_{\forall s} l_s, \quad (3.5)$$

where $T_{exec} = N_{cycles} T_{clk}$ is the duration of the application. Expression (3.5) is similar in form to (3.4) and also does not contain per-section activity. This shows that the form of (3.4) can be used as an approximation when leakage dominates, with a slightly modified technology constant.

Wire sectioning The type of sectioning switch that will serve as our workhorse: the 6-way 3-terminal (6W3T) buffering switch, is drawn in

[1] Such assumption is used by [13], and also regularly by TAD and other IMEC researchers.

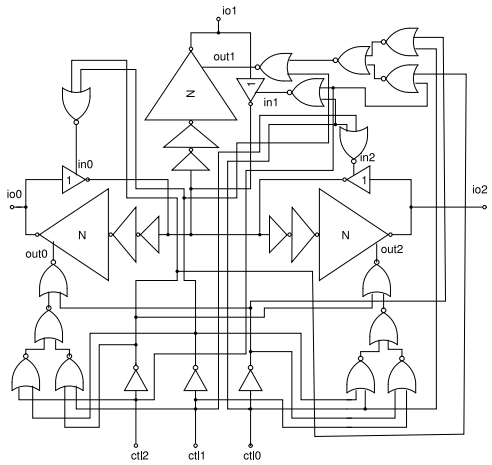


Figure 3.2: Six-way three-terminal switch (6W3T) with buffers of staged size. The size of the buffers is determined by data-plane design. A single decoder is used for a word-wide switch. The control decode circuit in the figure is of default type: it receives a control code which is 3 bits wide. In our design framework, non-default decode circuits can have more control wires and larger decode circuitry.

Fig. 3.2. It has staged-size output drivers and minimal-size inputs, and proves for our purpose versatile enough. It has a simple decode circuit for control of all 7 states (6 ways, plus disconnected state) by means of 3 control inputs (*ctl2*, *ctl1*, *ctl0*). Usually the decoder is more complex; the type shown is the default (and reference) type.

Using switches, we consistently divide all wires in sections. The switches are also repeaters. Switches are placed wherever three wires meet. In design, repeaters are optimally sized and separated to minimize the interconnect delay. Banerjee [13] and Wong [121] show that considerable freedom exists in placement of the buffers; we can therefore place them at wire junctions.[2]

Observing Fig. 3.2, it could be thought that the area and power dissipation of the switches is large. In fact, only the area of the buffer stages is appreciable, while the decoder's gates are minimum-size. Moreover, in multiple switches used for parallel wire buses of

[2]More repeaters may be present than sectioning switches, placed mid wire-section. They can be assimilated to 2-way 2-terminal (2W2T) switches, and must also be driven from the control plane. This does not increase topological complexity, and can be easily accommodated by our design framework. Such 2W2T switches are not mentioned any more below.

word size, the decoding circuitry is shared by the wires. The area of optimally-sized repeaters can be as high as 450 times minimum-size (cfr. [14], albeit for global-tier, not medium-distance lines). We do not count the buffers' area of the switches as control costs, since they replace repeaters that would be present anyway. It is exactly the cost of these repeaters which is represented by transport energy losses and minimized by EESC.

The advantage of wire sectioning is made clear by (3.3). If we can limit the sum $\sum_{\forall s} \alpha_s l_s$ in (3.3) to those sections needed for information processing, the total E_{xp} is strongly reduced. If we also choose l_s for minimal $\alpha_s l_s$ on those same wire sections, the effect will even be more pronounced. This was already illustrated graphically before, in Fig. 1.6.

Gating principle Our gating principle ("power gating for wires") bears some resemblance to power or supply voltage gating, an established circuit design technique for reducing (static) leakage power dissipation [95]. With power gating for circuitry, supply voltage is turned off whenever a circuit is not in use. With EESC, we do similarly for the buffers that drive wire sections. Because of power gating, the sum in (3.3) is only over those cycles and sections that truly contribute to information processing. This is optimal for dynamic and also for static loss. In comparison to power gating for circuitry, we have the benefit of reducing both dynamic and static losses.

3.1.2 The Program-controlled Tile

In a SoC, communication takes place between communicating entities called *terminals*, over channels which are wires, within the range of a *program-controlled tile*. We identify the entities in this domain, their structure, what resources the networks contain, and how they are partitioned. We review the concept of bandwidth which is known for terminals but has to be adapted to serve for communication architectures as a whole.

Minimum and maximum range The distance range of a program-controlled domain must be large enough for energy consumption by wires to matter, and small enough for synchronous operation. According to a reasoning by T. Noll [85], the minimum distance we need to consider is where wire-driven delay takes over from gate-driven delay:

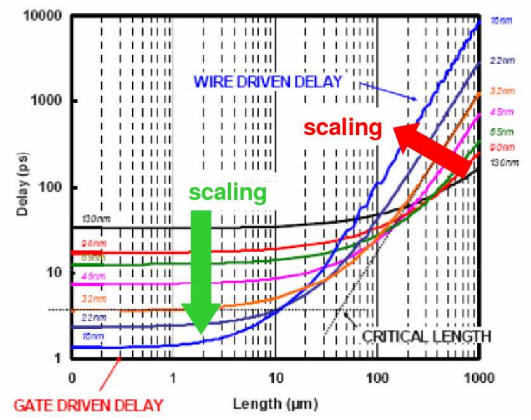


Figure 3.3: Critical lengths for different technology nodes. (Graph by T. Noll, [85])

this is about 2×10^3 transistor feature sizes, as shown in Fig. 3.3. The maximum is the distance reachable in a single clock cycle. This is almost constant in multiples of feature length; the value is 10^5 [63]. For a 45 nm feature size, the bounds on range are $90\mu m - 4.5mm$.

Modules and ports, terminals and terminal classes Within program-controlled domains, we find certain types of *modules* that exchange data: load/store units, functional units, register files, pipeline registers, parts of memory hierarchies (scratchpads and caches), bridges to external communication systems. Some of the modules may have multiple ports. The assignment (binding) of communications to ports is done design-time: when communication instructions are decoded, the ISA is explicit as to which ports of a module need to be used. Ports on the same module do not communicate, at least not over the network. Every port of a module is a terminal. Terminals of the same type, e.g. of different memories or corresponding ports of functional units, form a *terminal class*, which allow terminal arrangements to be formulated.

Terminal arrangements A terminal arrangement describes which terminals communicate, thus have transfers between them. Requirements for terminal arrangements follow from the processor's ISA, and are often expressed in terms of terminal class membership. Terminals inside SoCs tend to be arranged like shown in Fig. 3.4, where the ISA requires a certain number of concurrent transfers between any pair of members

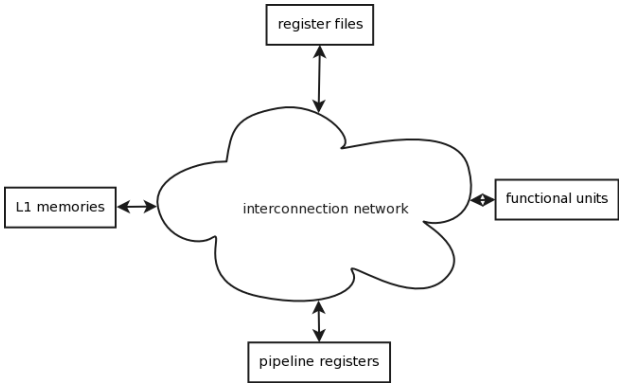


Figure 3.4: A terminal arrangement with four terminal classes.

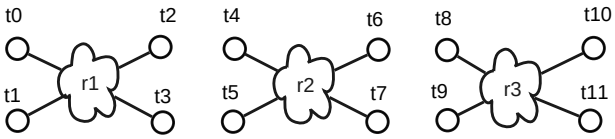


Figure 3.5: An external view of a composite network.

of 4 terminal classes: level-1 (L1) memories, register files, pipeline registers and functional units. Since an ISA is a language construct and has a free grammar, the type of constraint it imposes on the terminal arrangement can vary. Establishing the terminal arrangement is always the first step in deciding the sets of concurrent transfers imposed by the ISA. Inter-tile communication often has an *"all-to-all" terminal arrangement*, like, for instance, when the terminals are fat computing nodes; within SoCs, this is often not the case.

Resource network In our scheme, communication *resources* are switches or multiplexers. Wire sections are not considered to be resources, since they cannot themselves be controlled. The resource network has resources for vertices and wire sections for edges. The topology formed by terminals and resources, together with the set of paths required by the ISA, informally make up the communication architecture (CA). We can see this as a combination of physical and logical connectivity: physical connectivity is provided by the network, logical structure by the terminal arrangement; the formal definition comes later.

Composite networks The resource network may have disconnected parts; then we call it *composite*. A composite network, like that shown in Fig. 3.5, results in a composite CA, with disconnected *component CAs*. In the figure, component networks *r1*, *r2* and *r3* do not allow transfers between the terminals of each distinct component.

Bandwidth Bandwidth is the maximum number of transfers per cycle time. Bandwidth is commonly attributed to a terminal, or to an architecture with single in- and output. For multi-terminal architectures, bandwidth refers to the maximum number of *concurrent* transfers. This poses no problem for shared-media, where the bandwidth is 1 transfer per cycle, or point-to-point architectures, where transfers are decoupled. In other cases, we must be careful. Bandwidth calculations are discussed in Chapter 4. Here we note that the total communication architecture bandwidth is the sum of the bandwidths of the components. Components can then be *(i)* shared-media and allow a single concurrent transfer per cycle, or *(ii)* direct ("point-to-point") and allow a fixed number of concurrent transfers (which is not likely to be our choice, because of the inherent lack of scalability), or *(iii)* indirect and allow a variable number of concurrent transfers. If any of the component CAs has variable bandwidth, the whole communication architecture has variable bandwidth.

Whether or not the CA's bandwidth is variable, if it has N terminals with peak bandwidth B_T, the following constraint applies to the communication architecture bandwidth B_{A_C}:

$$B_{A_C} \leq \binom{N}{2} B_T = \frac{N!}{(N-2)!2!} B_T \tag{3.6}$$

This follows from our definition of bandwidth and of a terminal, which can at most communicate once per cycle. If the terminal arrangement is more limited than all-to-all, the constraint will be stronger. The constraint is external to network topology, which itself will impose more constraints.

Example communication architecture As an example, we consider a CA with a single component network and a linear topology. In Fig. 3.6, the CA has been used in a processor within a hierarchy of 8 scratchpad memories. The network is subdivided into a data network and a memory address network. Both are driven by the same set of control sig-

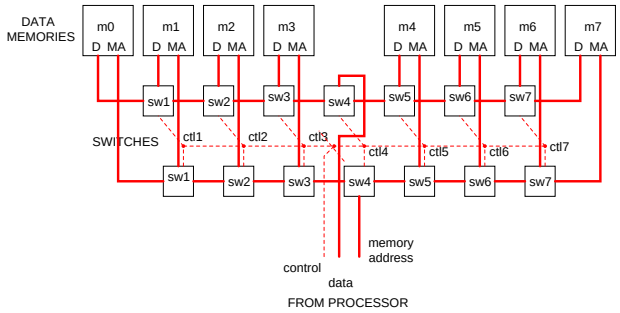


Figure 3.6: A template for a single-component CA with 8 scratchpad memories and sectioned linear network topology The CA is driven from the load/-store unit of the processor. In this example, both data and address buses are sectioned.

nals. The terminal classes are the processor's load/store unit (a terminal class with only one member) and the set of memories. The terminal arrangement prescribes a single transfer (read or write) between two distinct terminal classes, and not, for instance, a memory-to-memory transfer. The terminal arrangement and resource network are shown in Fig. 3.7. The number N of terminals is 9.

Although the network topology may allow many more transfers (it is "non-blocking"), the terminal arrangement can impose limitations that make it "blocking", and the communication architecture bandwidth fixed. In this example, although the network does not block a transfer from terminal `m5` to `m7` while a transfer from processor to `m3` is ongoing, the terminal arrangement does. The CA bandwidth is $B_{A_C} = B_T$, instead of variable and $\leq \binom{2}{9} B_T$, irrespective of the constraints that topology imposes, not as yet analyzed.

3.2 Programmed Control

In this section, we qualify our choice for programmed control of EESC. As shown, issues arise of determining what scheduling is, when to do it, and how difficult it is, especially for variable-bandwidth architectures.

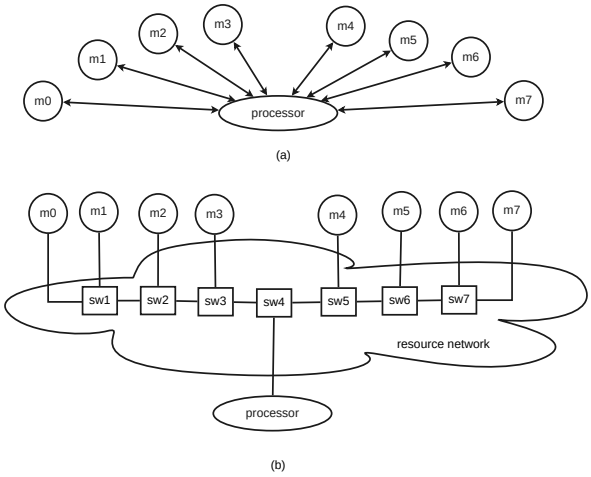


Figure 3.7: Terminal arrangement (a) and resource network (b) for the communication architecture of Fig. 3.6.

3.2.1 Program Control Flow

Program control flow is implemented by conditional instructions in program source code. In SoCs, programming is indispensable for the complex control flow inherent to today's applications. Controlling EESC is at least as complex, in terms of control flow, as the control flow of the program that drives the network. Our idea is to use program control flow itself to control the EESC network. Inspiration was provided by a 1953 paper [119] by Wilkes and Stringer.[3] Wilkes advocated programmed control over all resources of a programmed data processing machine, rather than hard-wired control by logic mechanisms. His argument is that programmed control scales better with size of the machine than hard-wired control, since it re-uses the control structures found in the program. With EESC, control decisions made once before are re-used because they are embedded in the control flow of the program.

When discriminating between programmed and hard-wired control, one should not fall into the fallacy that, because a hard condition generates an event, this event is by necessity hard-wired. After all, the *whole* processor is, at the bottom architectural level, activated by sig-

[3]This paper deeply influenced the semiconductor industry in the seventies and eighties of the last century. It spawned the concept of the micro-programmed processor, which subsequently became ubiquitous for more than a decade.

nals. It is the presence of a control flow graph, encoded in branching instructions, that makes a difference. A process is under programmed control if at any level of its control flow, the occurrence of events is determined by a program written in a sequence of instructions, having the semantics of a programming language. The question "where is the program?" usually settles the issue.

Programmed control should also not be confused with software-assisted design of elements of control. We try to avoid the term "software control", since it does not exclude the case where software is used at design time, but the controlled machine is not programmable at run time. Such a machine does not pass the crucial test mentioned above, and neither does it follow Wilkes' prescription.

3.2.2 Static Scheduling

Since the act of scheduling is crucial, it makes sense to ask ourselves when it is performed: at design, compile or run time? Controlling the resources of a communication system can be seen to consist of four tasks. Using terminology from [82], they are: *(i)* identifying the resources to be controlled, *(ii)* partitioning the resources in sets that are controlled together, *(iii)* scheduling in the time domain, i.e. allocating transfers to time slots, and *(iv)* binding in the space domain: allocating resources to transfers. Chapter 2 has revealed that considerable variation exists in the approach to all these tasks. In our concept, the first two tasks take place at design time; the latter two, preferably at compile time, or else at run time.

We call this *static scheduling*. It is by no means a universal principle, but common enough in the domain of embedded processors, including very-long-instruction words (VLIWs), to serve as the guideline of the design framework of this thesis.[4]

Identification of resources Resources can be identified at different levels of abstraction: wire systems, routes, arbitrators, buffers, multiplexers, switches. Consistent with our aim of control at the physical level, we will always choose the lowest level of abstraction. To control

[4]Superscalar processors, including most general-purpose (Intel x86) and some (ARM) embedded processors, do not follow the static scheduling paradigm. This does not wholly disqualify our design framework for such processors, but raises extra issues which we do not treat in this work.

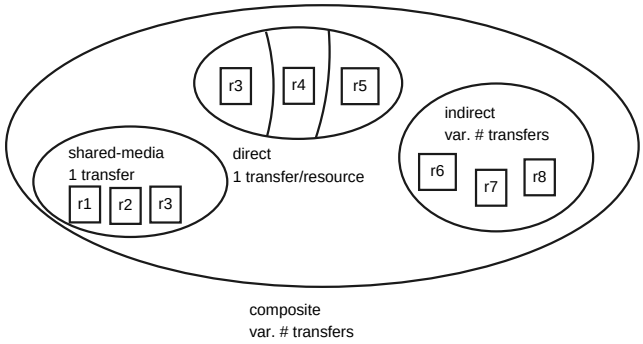


Figure 3.8: A composite communication architecture may have shared-media, direct and indirect components. The total number of transfers to be scheduled depends on the bandwidth of each component.

something, it must be controllable within its most inner state.

Partitioning resources Scheduling does not always apply to individual resources, but sometimes to parts of the resource group. A key question with resource partitioning is how many transfers can/should be scheduled per scheduling cycle. Partitioning of resources ties in with the distinction between shared-media, direct, and indirect networks.

Multiple networks are disconnected and have no common resources: they can be controlled separately and form a partition at top level. Hierarchical networks have a tree-like resource partitioning. Shared-media networks have no partitioning at all: all resources are consumed by any single transfer. Direct networks have one resource per transfer. Each transfer consumes a single resource, and there can be a fixed number of transfers per cycle. Indirect networks, where transfers are inter-related by conflicting use of resources, have a variable number of transfers per scheduling cycle.

The bandwidth of a communication system (”architecture”) is a result of resource partitioning hierarchy. The bandwidth of a composite communication architecture is the sum of the bandwidth of its components. As seen from Fig. 3.8, if any component has variable bandwidth, the total communication architecture bandwidth is variable as well. Communication architectures with only shared-media or direct-network components are *fixed-bandwidth*: the first, because there is only one transfer per cycle, and the second because there are as many transfers per cycle as there are resources. If the bandwidth is fixed at N

transfers per cycle, a scheduler can choose to schedule any combination from 0 up to N transfers in any cycle to do this. If the bandwidth is not fixed, a component is *variable-bandwidth*. The number of transfers that can be scheduled in a cycle depends on the very set chosen. There may be a maximum number of possible transfers per cycle. We have not yet the mathematical concepts to decide this, but we do know that property of an architecture having fixed or variable bandwidth is inherited from the component by the architecture as a whole.

Scheduling Scheduling is the decision of how many, and which, transfers will make use of each time slot. Scheduling is easy for a fixed-bandwidth architecture, less so when bandwidth is variable. In variable-bandwidth systems, dependency exists between transfers that may or may not be scheduled concurrently, depending on the resources consumed by each. The scheduler really needs to know details of the communication architecture. If he does, he can still make a decision, taking all run time possibilities in account. With communication architectures, the act of scheduling includes an aspect of *routing*. A routing table can have been prepared at design time; with program control, under the assumption of static scheduling, this is always the case.

Binding Binding is the allocation of resources to transfers. Binding, in a programmed system, can be done at compile time, but not always completely. Indeed, a conventional compiler cannot fully resolve the terminals that will be communicating. An amount of run time binding of resources may have to be done at run time, even under programmed control. This is for instance the case when the terminal is a memory bank selected on the basis of addresses known only at run time.

3.2.3 Definition of Programmed Control

We call a communication architecture *program-controlled* when *(i)* there exists a program whose flow controls the architecture; *(ii)* all scheduling is implied in the instructions of the program; *(iii)* the scheduling decisions are made compile time; *(iv)* binding is done on the basis of instructions, as far as possible. This means that all information available to the compiler is used in binding, even if the binding is incomplete. It implies that residual binding is performed at run time.

Exceptions in scheduling (interrupts, cache misses,[5] direct memory access (DMA), transfers to other, asynchronous, domains) are handled by deterministic control procedures. These are synchronous to and coordinated with the program control flow, thus they are compatible with programmed control, even if their controllers are hard-wired.

3.2.4 The Communication Processor

The communication processor (CP) is a paradigm that guides us throughout our evaluation of program-controlled EESC, keeps us from straying into misconceptions, and helps us to cast the actors of our play: the designer, compiler (including its back end, the assembler), the program and the programmed machine. The intention of the paradigm is to be simple, compatible with many types of embedded VLIWs, and to allow programmed control to be investigated, simulated, demonstrated, and optimized.

Components of the communication processor The communication-specific parts of the CP are shown in Fig. 3.9. The *path decoder's* role in the communication processor is comparable, in a way, to that of the ALU in the main processor; only it handles topology, not arithmetic or logic. In another way, it is an extension of the instruction decoder already present. Its functionality depends on the way use of resources has been pre-encoded in the ISA. It converts communication instructions to switch control bits. To do this, it needs topological knowledge of the data network. *Control processing* in Fig. 3.9 refers to components[6] introduced specifically to reduce control costs, like splitters of control or transport loop buffers. The networks – drawn in Fig. 3.9 as "clouds" – consist of a data network[7] and a control network – two clouds superimposed. Vertices in the data network correspond one-to-one to ver-

[5] IMEC design style, using DTSE, favors scratchpad memories above caches. Use of caches, even with a hard-wired cache controller, is not incompatible with programmed control elsewhere on the tile.

[6] We do not include registering in control processing, i.e. using memories specifically for control information, since we concentrate on frequently reconfigured networks. Registering requires extra costs for access, and becomes more feasible the less frequent the reconfiguration is.

[7] The direct link between the communication instruction register and the data network in Fig. 3.9 is a detail, included for completeness. It allows for the possibility of immediate addressing (see below). Some part of the communication instruction can be injected straight into the data plane.

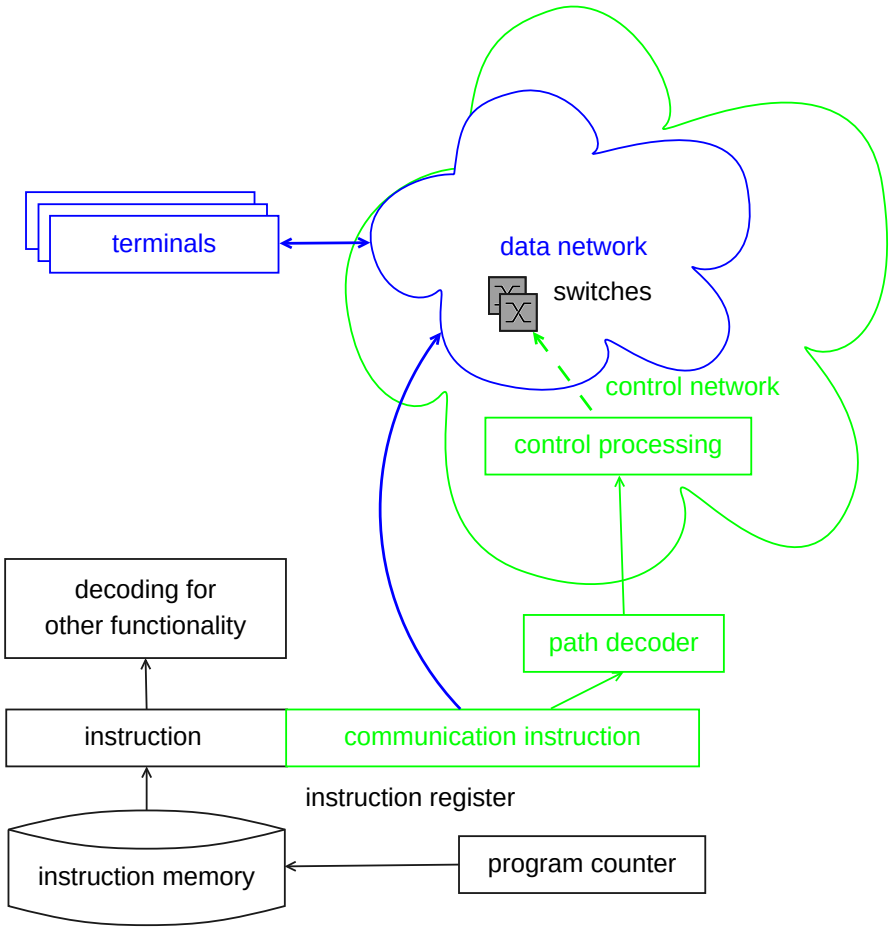


Figure 3.9: The communication processor paradigm.

tices in the control network. After all, every data switch needs control as well.

The data plane With respect to EESC, memory or register address networks are just a part of the data plane. For instance, the data plane in Fig. 3.6, our linear-topology example communication architecture, contained an address as well as a data network. If an address network has the same topology as the data network, we use the same control codes for both. In [59], we found that sectioning gain in a memory address network can be as high as that in the data network, at almost

no extra cost in control.

Access protocols Transporting data of addressed type in the data plane poses specific feasibility problems. Data cannot be accessed without at least a simple access protocol, specifying the timing of address and data phases. Such a protocol can be improved by techniques like data/address pipelining, sequential and split transfers. To incorporate these in the control plane is cheap and effective. To verify their correct operation is not trivial. Simulations made in preparation for the publication of [59], convinced us that a simple protocol could work. Further than those, we considered optimization of address/data timing out of scope for the control plane.

Importance of the instruction set architecture At the instruction level of the machine, the use of a communication processor presumes the definition of elements of a *communication instruction set*. Our aim is to configure the sectioning switches for every instruction cycle that includes a transfer. A program can do this, provided the necessary features are included in the ISA. The part of the ISA pertaining to communication is the *communication ISA*. Likewise, the part of the instruction relating to communication is called the *communication instruction*. Machine language instructions must include features that allow the desired network state to be set, i.e. the resources to be used during each transfer.

We call an ISA *transfer-aware* if some transfers between modules (terminals) can be controlled from a program. This is in fact always the case: all instruction sets (ISs) define some load/store transfers, even if some other transfers are invisible to the IS (for example, instructions transferring addresses to the program counter). Communication instructions comprise all features of the instruction that initiate a transfer in the CA. If functional units of the machine are connected by the CA, the operational instructions (add, multiply ...) are in part communication instructions. This is also transfer-awareness of the ISA.

If any other properties of the intra-processor communication system than transfers are exposed to the IS, we call the processor and its ISA *communication-aware*. For example, the transport loop buffer control instructions employed in [59] are communication-aware: they exist solely for control processing in the communication architecture. Communication awareness might be implemented for various reasons,

likely having to do with resource conservation: energy efficiency, congestion avoidance, reconfiguration in the presence of device errors or variability. It is a interesting property, since few contemporary embedded processors display it, as seen from a recent survey like [86].

A special form of communication-awareness is awareness of topology. We call an ISA *topology-aware* if the IS is able to control the paths followed by intra-processor communication, not only the endpoints. Topology-awareness is not common, although it might become so with future requirements for the data plane.

3.2.5 Scalability of Control: Principle of Minimal Width

Fixed-bandwidth architectures are trivial to control, even with many terminals. There are various reasons why architectures may become more complex than shared-media or point-to-point. Mesh topologies allow multiple concurrent transfers, but are variable-bandwidth. Hierarchical CAs need to be variable-bandwidth if their resources are to be fully exploited: when one part of the hierarchy is used for a transfer, another part is still available for use. Bus components that are basically fixed-bandwidth may become variable-bandwidth because of occasional interconnections. Even linear topologies can be used in a variable-bandwidth way, boosting resource utilization, as we shall see later.

Scaling with complexity As network complexity increases, when and why does EESC become uncontrollable? Not by failure to gain from wire sectioning: a large complex data network profits proportionally to the number of unused sections. If relatively more sections of the network are not driven during some transfer cycles, sectioning gain increases. To understand the scalability problem, we must watch the flow of control information, shown in Fig. 3.10. Control originates from the fetching and decoding of the communication instruction, the communication-aware part of the instruction set. The path decoder recodes this information according to its knowledge of the CA. Control processing operates on the control information before or while it is distributed over the surface of the tile. Ultimately, every switch must be provided with a control code for every cycle. This holds true even if network resources are organized hierarchically: hierarchical organizations reduce the amount of control information, but the control infor-

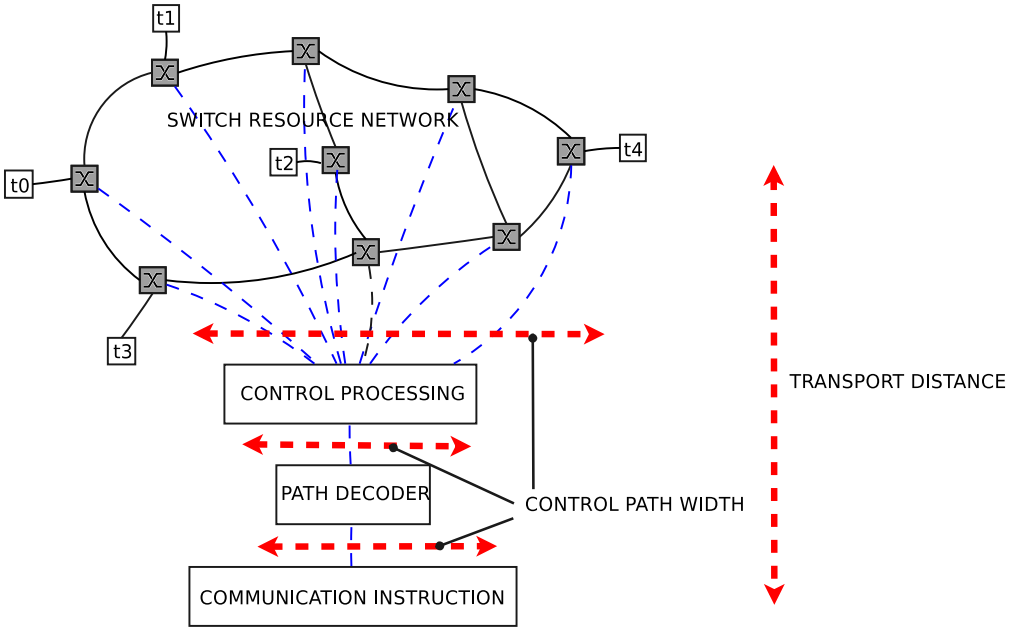


Figure 3.10: The width of the control path varies at different sections of the control plane of the communication processor. At each stage (after instruction decoding, path decoding, and before distribution to the switches), width is related to costs of control.

mation must cover approximately the same distance as the data, follow approximately the same path, since it is not registered.

Communication instruction width is expensive, since it consumes a part of the general budget for instruction fetching. Transporting switch control codes over many parallel wires is also costly, not only in chip area but also in power and wire congestion. If the activities and lengths of control wires are of the same order of those of data wires, the transport energy required to distribute control will become comparable to the energy won by EESC, once the bitwidth of the control plane is comparable.

We suspect that control efficiency breaks when the control path becomes too wide; our use cases will confirm this. With too many switches to control, the EESC principle breaks. This eventuality guides our optimization of control: we will be looking for *minimal width* at every section of the control plane: while fetching and decoding instructions, doing path (trans)coding and distributing control codes.

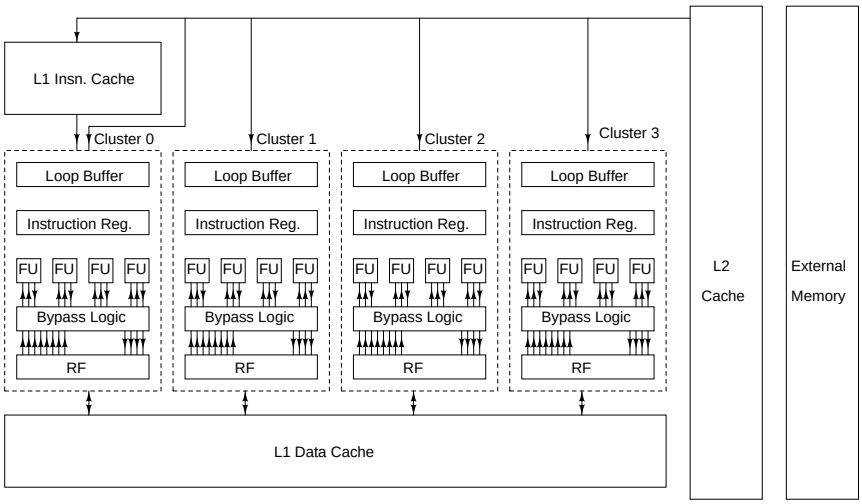


Figure 3.11: An instance of CRISP (RF: Register File, FU: Functional Unit) Source: [15].

3.3 State of the Art for Programmed Control

Having narrowed down the concept of programmed control over EESC, we turn to literature to see which processors might use similar concepts, and where our own concept of EESC control might fit most readily. Under assumption of static scheduling and frequent reconfiguration, the most suitable way to decide which class of communication-awareness a processor belongs to, is to inspect the ISA. Indeed, any feature not present in the ISA is ipso facto not controllable by a program. We will determine whether there is programmed control, whether we have fixed or variable bandwidth, and which features of the intra-processor network are controlled, i.e. exposed to the ISA.

The CRISP processor CRISP (Configurable and Reconfigurable Instruction Set Processor) is a VLIW research processor developed at ESAT, Katholieke Universiteit Leuven, and IMEC. It was the subject of 3 theses [15, 68, 111] at IMEC. It has the following outstanding features. The ISA is built upon the HPL-PD ISA. Instances of CRISP are configured by means of XPML, an architecture description language for high-level design space exploration, based on Extended Markup Language (XML). Configurable features include data path components, like number of clusters, number of instruction issue slots per

cluster, functional units and register files, data and instruction memory hierarchies. It has a data-path bypass network allowing to chain functional units within a clock cycle. The instruction fetch architecture is scalable, designed for low power consumption and features loop buffers. The design space exploration framework consists of a compiler and a simulator, derived from the Trimaran framework [110] and a power estimation tool.

A diagram of an instance of CRISP is shown in Fig. 3.11, where four separate instruction-issue clusters have been implemented. In our terms, CRISP is a transport-aware processor, with the peculiarity that the transfers include broadcast transfers. The intra-tile network centers around the configurable functional units, register files and load-/store units found in each cluster. The communication architecture's bandwidth is fixed, since the compiler includes no facility to schedule a variable number of transfers per cycle. There is no topology awareness in the ISA. Frequent reconfiguration and static scheduling apply. Chapter 8 contains an example of control of EESC for CRISP.

Transport-triggered Architecture (TTA) The key idea in TTAs, another class of VLIW architectures, is to code transfers between functional units within the instructions, not functions and their operands. The transfers to the local operand registers of the functions automatically trigger function execution. There are, in essence, no other instructions than transfer instructions, making a TTA a transfer-aware processor *par excellence*. Dependencies and independencies are resolved at compile time. TTAs are fixed-bandwidth, and to our knowledge make no use of other topologies than "buses" (multiple shared-media), even in recent implementations such as the TACO Internet Protocol (IPv6) Processor [112]. This leads to our suggestion that TTAs could benefit from variable bandwidth and the more fine-grained communication control proposed in this work for EESC.

The Imagine architecture The Imagine chip is a stream processor that contains 8 clusters of arithmetic units (tiles, in our parlance) processing data from a stream register file (SRF). The SRF, a large on-chip storage for streams, is the nexus of connectivity between tiles. It is, in effect, an inter-tile communication system, intended to carry 8 payloads of stream type. The communication bandwidth is fixed. Within the tiles, the communication system is a buffered crossbar, which in itself is not

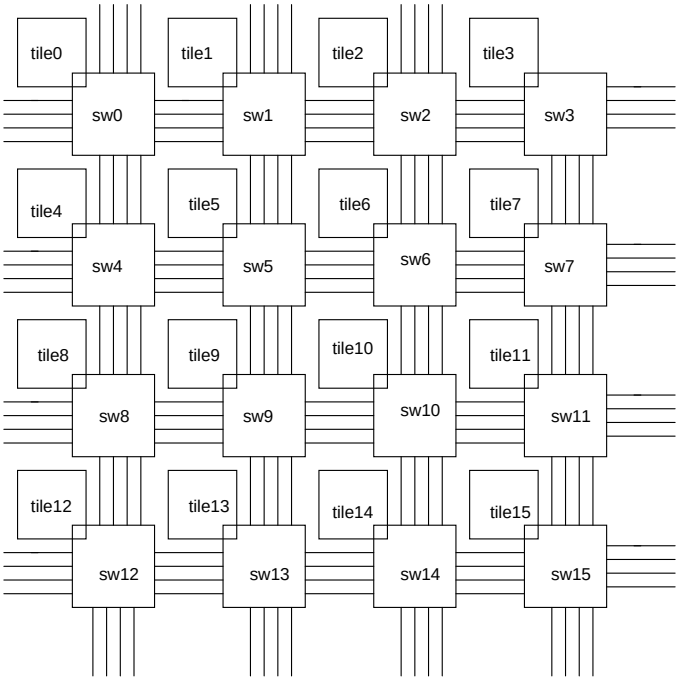


Figure 3.12: The Raw processor comprises 16 tiles. The tiles are "fat nodes": they contain compute resources, memories and a router.

an optimized structure for low-power consumption. We conclude that also this computer is only transfer-aware. Since its communication network is special-purpose, it is not immediately clear how much could be contributed by programmed control of EESC.

The Raw processor The Raw processor's design philosophy explicitly intends "to bare all to software", including pins and wires of the on-chip networks. Every aspect of communication is program-controlled, including topology.

Considering multicore processors like Raw, we must separate the intra-tile network from the inter-tile network. Since Raw was not specifically designed as a low-power architecture, the intra-tile network has no sectioned communication. The inter-tile network (depicted in Fig. 3.12) consists of two static networks, for which the routes are specified at compile time, and two dynamic networks, where routing is performed at run time. Both networks (intra- and inter-tile) are under

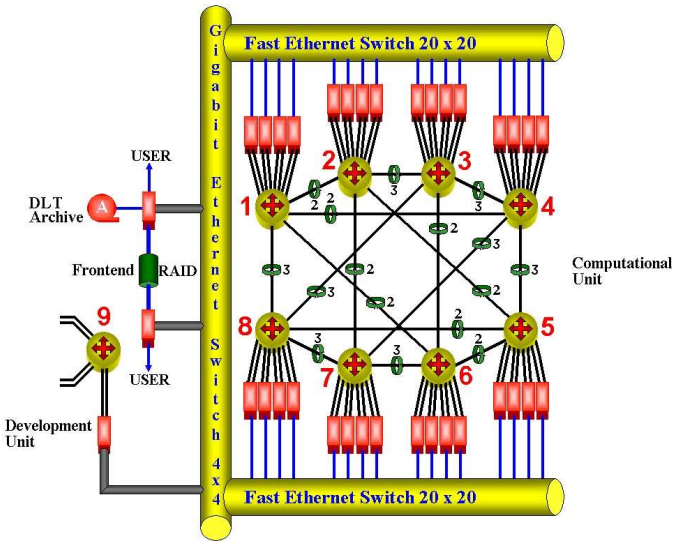


Figure 3.13: The network of the Swiss T1 supercomputer. The numbers on the links of the central network architecture denote the number of concurrent transfers for all-to-all operation between the switches 1-8. (Source: [42]).

programmed control. The inter-tile scheduling by the compiler [39] is limited to the static networks; the dynamic networks are used for cache misses, interrupts and other asynchronous events.

The input ports of the switches each have a 4-element FIFO. Communication is thus store-and-forward, not circuit-oriented. The whole structure is synchronous: every wire is registered at the input to its destination tile. The longest delay in the system is no longer than the delay of a hop in the intra-tile networks. To the processor tiles, the inter-tile network appears as a register file. Values can be read off and put on the network with no overhead in latency. The programmable switches route operands through the network, as instructed by per-switch *route* instructions, which incidentally, are 64-bit wide.

Nurmi [86], chapter 14, gives an extensive account of the Raw architecture. For our part, we conclude that even inter-tile communication can be program-controlled in a topology-aware way. Since programmed control in a stream multicore processor is not what we envisaged in our work, a transposition of EESC to Raw would not be straightforward, although there are no fundamental incompatibilities.

The Swiss T1 supercomputer Supercomputing multiprocessors often blaze new trails in interconnection methodology. The Swiss T1 Supercomputer is a commodity supercomputer built from 70 Compaq Alpha processors. It features a central computational network with 8 12x12 crossbars switches, each connecting 4 dual-processor computers to a topology consisting of two superimposed K-rings, shown in Fig. 3.13.

The supercomputer as a whole does not form a single program-controlled domain, nor has it an overall ISA as such. The bandwidth is variable, the scheduling of communication is collective and topology-aware. Indeed the method of liquid scheduling described in [36] has some similarities to our theory of useful-state encoding, but the type of constraints imposed by the topology are different. In comparison to our methodology, the switches of [36] are not atomic, and the principle of optimality for scheduling is the maximal use of bottleneck links by a given set of transfers, whereas we consider every possible transfer set per cycle.

3.4 Scope of Program Control

In this chapter, we have explained how EESC reduces power consumption by interconnects; why programmed control creates a niche for EESC, for medium-distance communication in programmed intra-tile domains; we have presented the principle of static scheduling and the paradigm of the communication processor, and settled on a criterion of optimization for our design framework.

Scope of the communication processor paradigm The communication processor paradigm is applicable in the domain of *(i)* medium-distance (intra-tile) on-chip communication, for *(ii)* embedded systems-on-chips. under the assumptions of *(iii)* static scheduling, and *(iv)* frequently reconfiguration. These conditions cover academic processors like CRISP, DLX, Trimaran and VEX (all of which, except DLX, are research-type VLIWs); some versions of open-source processors like Leon, and of industrial embedded processors like ARM, MicroBlaze (an extension of DLX), NIOS II, PowerPC, SPARC, and Super-H, (some of which are IP cores for FPGAs). Our scope is thus not unlike that of IMEC's COFFEE architecture exploration framework [100, 101]. Even processors using some concepts similar to our own, however, may have

features that cannot be immediately accommodated. This does not reflect on the feasibility of EESC itself, nor on control of EESC. Rather, we conclude that programmed control of sectioned communication is an intrusive principle that permeates the entire design of a SoC, and has to be built-in from the start.

As follows from Section 3.1.1 and [50, 92], EESC is itself applicable to all deep sub-micron-technologies.[8] Our design pattern and framework, based on the communication processor (CP) paradigm is, like all TAD work, concerned with the interface to processing architectures, physical design and platforms, rather these fields themselves.

[8]For hard-wired circuits EESC can be controlled by a custom-built sequencer, having the basic functionality of a communication processor (CP). The reason why such a sequencer can be simple will be found in the next chapter on Useful-state Analysis and useful-state encoding.

Chapter 4

Network Control with Minimal Redundancy

What's the use of looking when you don't know what it means?

Elvis Costello, *Mystery Dance*.

This chapter describes the theory of useful-state analysis (USA) and useful-state encoding (USE). We start with a preparatory mapping of the subject. Using a simple example for illustration, we provide mathematical definitions for the entities of a communication architecture (CA), and for the sets of paths and transfers that traverse the network. We show that, in the context of a particular communication architecture, the state of a network can be described by a set of useful states smaller than the combination of all resource states. This description has minimum redundancy.

Useful-state analysis is the procedure of finding the useful states. To perform it, we define some types of graphs to be constructed from CA specifications and present an algorithm operating on the Path Allocation Graph (PAG).

Since our first example, used for definitions is oversimplified, we will also illustrate real-life USA with a second example. We will ultimately define communication architecture bandwidth, propose figures of merit for communication architectures, and end with with a summary of useful-state encoding, including a survey of the broader application of USE to frequently reconfigured transport networks.

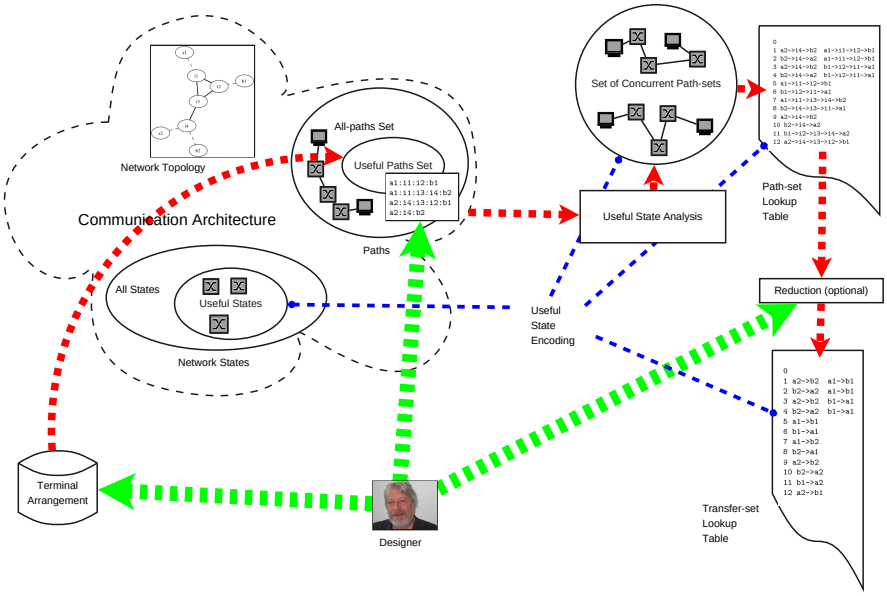


Figure 4.1: A mind map for the reader: communication architecture, useful-state analysis, useful-state encoding, path and transfer lookup tables.

COMMUNICATION architectures, defined later in this chapter, describe both physical and logical connectivity in a network. Intuitively they can be seen as a combination of network topology and terminal arrangement. If a communication architecture (CA) contains other components than shared-media and point-to-point networks, this raises issues in respect to transfer scheduling. We need a framework to handle the allocation of resources of CAs that can be complex, containing meshes, rings, trees, hierarchical or irregular topologies. Experiments with linear-topology CAs have led us to identify the primary problem with large topologies: the explosion in width of the control plane, due to inordinate growth of the control state space. Useful-state analysis will show that this growth is avoidable: much of the control state space has no use for actually realizing communicating paths. Often, the network's control state space can be limited to a size that is still manageable.

To explain resource allocation in CAs, we must first define a number of communication concepts in terms of graph theory. Then we will define the concept of network states and explain the advantage of a network description in terms of useful states. Ultimately we will arrive at

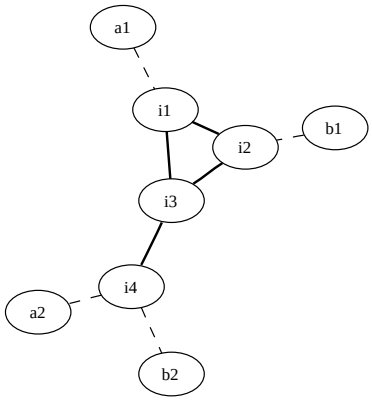
useful-state analysis, which allows us to find the useful state set. The concepts to be defined have been visualized in the mind map of Fig. 4.1. This allows the reader to situate every aspect in relation to the others; when every concept is fully defined, we will refer back to the map. The map features a *terminal arrangement*, a *communication architecture* consisting of a *network topology* and a *useful paths set*. The communication architecture has a number of possible states, but the *useful-state set* is only a subset of those. Useful-state Analysis is the procedure that yields the *set of concurrent path-sets* from the communication architecture's description, and assigns control codes to each of the sets. A concurrent path-set then corresponds one-to-one to a useful state. USA results in a *path-set lookup table*, which links control codes to concurrent path-sets and useful states. Optionally, a path-set lookup table can be *reduced* or just rewritten as a transfer-set lookup table. At various points in the procedure the designer intervenes in the procedure: Useful-state Analysis is not only mathematical analysis, but rather a design process. The designer must interpret the terminal arrangement, select the useful paths from the topology's all-paths set, and perform concurrent path-set reduction if he or she deems it necessary or opportune.

The work described in this chapter is wholly original, except where stated. Since the result is powerful but basically simple, it cannot be excluded that others have thought about this and expressed it in another form. In any case, the reference works on interconnection engineering [29, 31, 94] do not mention it.

4.1 Communication Architectures

In this section, we define communicating entities, network resources, communication architectures, and sets of paths and transfers. Fig. 4.2 shows the example of a simple communication system, a *communication architecture (CA)*, consisting of a *network graph* (on the left) and a *set of useful paths* (on the right). The network graph is sometimes called a *topology*. The example is not drawn from practice, but simple enough for the concepts to be readily pictured and understood.

Network graphs A *network graph* G_{NW} is a graph (V, E). It may be undirected or directed. The vertices V are terminals and resources. The edges E are wire sections. Undirected network graphs allow two-way communication on all wires. Directed network graphs model networks




```
a1:i1:i2:b1
a1:i1:i3:i4:b2
a2:i4:i3:i2:b1
a2:i4:b2
```

Figure 4.2: A communication architecture (*CA*): a network graph (on the left), with terminal vertices (`a1`, `a2`, `b1`, `b2`) and resource vertices (`i1`-`i4`), and (on the right) a set of useful paths between terminal vertices. The resource subnetwork is marked by bold edges.

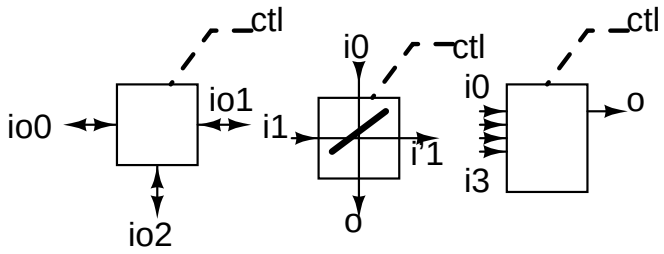


Figure 4.3: Control elements: 6W3T switch, crosspoint, multiplexer.

where at least some wire sections are one-way.

Terminals and resources *Terminals* are network vertices of degree 1. All other vertices are *resources*. The *resource network* is the subnetwork spanning the resources. In Fig. 4.2, its edges are marked in bold. It does not include the terminals and their drop-in sections, the single edges by which the terminals are connected. (Drop-in sections are marked as dashed edges.) The network vertices either belong to the *terminal set* $T_{NW} = \{t\}$, or to the *resource set* $R_{NW} = \{r\}$. In Fig. 4.2, $T(G_{NW}) = \{a1, a2, b1, b2\}$, and $R(G_{NW}) = \{i1, i2, i3, i4\}$. Resource networks with disconnected components belong to composite CAs, each of which can be analyzed separately. We will concentrate on connected resource networks, that make up a single component network.

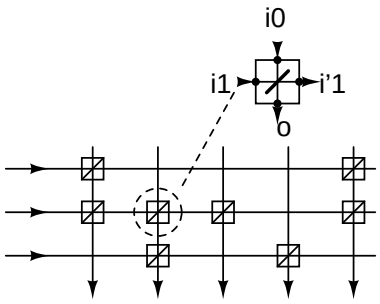


Figure 4.4: Crosspoints used in a multiple crossbar.

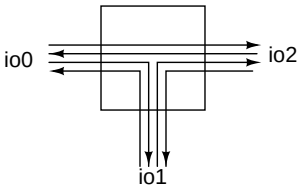


Figure 4.5: 6W3T control states: 6 ways of switching, plus disconnection.

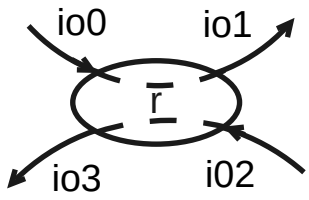


Figure 4.6: Forbidden switch state for a 4-terminal switch, with two paths.

Resources as control elements Resources can be switches, but also other types of control element. Among some choices, shown in Fig. 4.3, we see a 6W3T switch, a crosspoint[1] and a multiplexer. Our framework is applicable to all of them with little change. Multiplexers need an output disable control to be useful for wire sectioning.

A 6W3T switch, of the type used in the example network graph of Fig. 4.2, can be in one of 7 states, resulting from control: one state per way, plus one for total disconnection. These states are shown in Fig.4.5. A crosspoint has 3 states (a state relaying input *i0*, a state relaying *i1*, and the state of output disable). A 4-to-1 multiplexer with output disable has $4 + 1 = 5$ states.

Importantly, resources are *atomic*: they can only be allocated to a single path. If they are set up to realize a path, they can and should

[1] A crosspoint has 2 inputs *i0* and *i1*, the latter of which is looped through via *i'1* to the next crosspoint in a crossbar. This allows multiple crossbars to be constructed like in Fig. 4.4.

not realize another path simultaneously. For instance, a switch which has states corresponding to the paths depicted in Fig. 4.6 is not atomic. In that figure, a path enters the resource via port io_0 and leaves via port io_1, while another path enters via port io_2 and leaves via port io_3. Atomic resources do not allow such behavior. If this needs to be modeled, we should split up resources.

If necessary, non-atomic resources may be replaced with multiple atomic ones. In on-chip communication, close to the physical structure of the network, we are never far away from the atomic level, which corresponds to the actions of single gates and transistors. Splitting up non-atomic switches in atomic ones is always possible, but sometimes leads to using a different types of control element, or having to consider related control between elements. This is not an essential problem, and can be accounted for by bookkeeping and combinatorics. It complicates the formalism, and in this chapter we will not consider such situations.

That control states can be related could already be seen from Fig. 4.4, where the control of the individual crosspoints in a crossbar is correlated. A crossbar is often depicted as having "radio buttons": selecting one "button" releases the others. If control of the switchpoints is correlated, or if the switches are not of the same type, the states of a number of switches is not the product of the number of states of each switch.

The *resource state set* $S_{\mathfrak{r}}(r) = \{\mathfrak{r}\}$ is the set of states associated with a resource r. As stated above, we will usually assume all resources to be of the same type, thus all resource state sets to be identical. This is not essential to the theory, but it simplifies counting states.

Control code lookup A control code corresponds with every state of a switch. A *control code set* $S_{\mathfrak{c}}(r)$ is the set of control codes $\mathfrak{c}$, associated with a resource r. A *switch control code lookup table* is a one-to-one mapping from control code set $S_{\mathfrak{c}}(r)$ to resource state set $S_{\mathfrak{r}}(r)$, where the sizes of the sets $|S_{\mathfrak{c}}| = |S_{\mathfrak{r}}|$. Such a mapping can be implemented in hardware as a combinatorial circuit, or in hardware design language as a lookup table. In both cases, the map is implemented as a *decoder*. The 6W3T switch shown earlier, in Fig. 3.2, had a decoder of default type for such switches. Although resources are usually of same type, each normally has its own lookup table, thus a different decoder.

For all resource types, the number of wires needed to deliver control is $\lceil \log_2(|S_{\mathfrak{r}}|) \rceil$. For instance, for a 6W3T switch, $\lceil \log_2(7) \rceil = 3$.

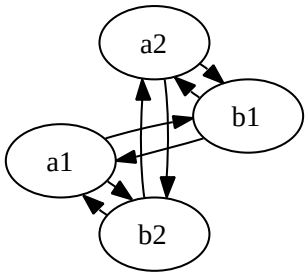


Figure 4.7: Terminal arrangement for the simple CA of Fig. 4.2.

Paths In a network graph, acyclic inter-terminal paths $p_{i,j} = (t_i, \ldots, t_j)$ can be realized. For our purpose, a *path* is always **acyclic**, because cyclic inter-terminal paths would contain some resources twice, contrary to our assumption of atomicity and thus single-path allocation. Paths can be directed or undirected. In the latter case, they are equivalent to two directed paths in anti-parallel.

The *all-paths set* $S_{AP}(G_{NW})$ of the network graph is the set of all acyclic paths between terminals. Finding all paths in a network graph is trivial for acyclic but not for meshed topologies. In the latter case, the designer can make use of the path-finding algorithms of Section 5.1.

A *transfer* $\mathfrak{t}$ is formally an ordered pair covered by a path: $\mathfrak{t} = (t_i, t_j)$ for which a path $p_{i,j} = (t_i, \ldots, t_j)$ or a path $p_{j,i} = (t_j, \ldots, t_i)$ exists. (Recall that paths may be undirected, so we must account for both possibilities $(t_i, \ldots, t_j)$ and $(t_j, \ldots, t_i)$.)

Terminal arrangement A *terminal arrangement* is a set of terminal constraints best expressed in a language; natural language has served us well. It can often be depicted as a directed simple graph $A_T = (V, E)$. The vertices $V(A_T)$ are terminals. The edges $E(A_T)$ are transfers. Fig. 4.7 shows the terminal arrangement for our example. The only transfers are between distinct terminal classes, not within a terminal class. The terminal classes are $A = \{a1, a2\}$ and $B = \{b1, b2\}$.

Communication architecture We can now give a formal definition of a *communication architecture (CA)*. It is a pair $A_C = (G_{NW}, S_{UP})$ of a network graph and a useful path-set. The network graph is the topology, but the useful path-set is chosen by the designer.

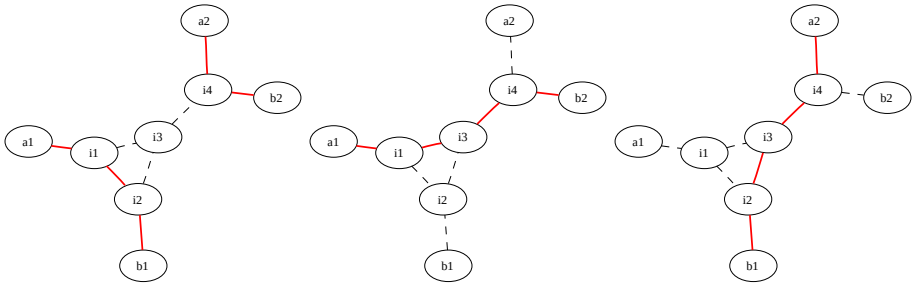


Figure 4.8: A set of concurrent path-sets. In each path-set, concurrent paths are marked by solid edges.

Useful-paths set The *useful-paths set* $S_{UP}(A_C)$ of the CA is a set of acyclic paths between terminals. It is a subset of the all-paths set S_{AP} of the network graph $G_{NW}(A_C)$: $S_{UP}(A_C) \subset S_{AP}(G_{NW}(A_C))$. Choosing the useful path-set is part of the design: in the useful path-set, only paths are selected that *(a)* will cover the required terminal arrangement, *(b)* are minimal, meaning that no other path in S_{UP} contains it, and *(c)* do not contain an excessive number of resources, in the judgment of the designer. If it is evident that some long detour consumes resources without adding usable redundancy, the designer can drop it. Else, the designer can be prudent, keeping the path at this point, and remove it later when reducing the path-set lookup table (PSLT) as described below.

Concurrent path-sets and set of concurrent path sets *Concurrency* for paths means that they are *resource-disjoint*, so that they can occur together. Fig. 4.8(a) shows such a *concurrent path-set (CPS)* for our example CA: a set of useful paths with no resources in common. The paths of a set are indicated in bold. The three sets in Fig. 4.8 form a set of concurrent path-sets for our example CA. The complete *set of concurrent path-sets (SCPS)* is termed S_{CPS}, the set of all CPSs. Whether a set of concurrent path-sets like that of Fig. 4.8 is complete, cannot be decided yet in all but trivial cases: it requires useful-state analysis, which we will come to.

Broadcasting In a *broadcast* architecture, single terminals can transfer simultaneously to all or some members of another terminal class (or their own class). For a broadcast CA, the concept of a useful path-set $S_{UP}(A_C)$ is replaced by that of a *useful path tree set* $S_{UPT}(A_C)$. It contains broadcast trees covering the terminal broadcasting arrangement. Useful-state analysis for broadcasting CAs can be performed in terms of *broadcast tree broadcast tree sets* instead of path-sets. Atomicity of switches with broadcasting means that resources can only be attributed to a single tree. We will not further mention specific notation or terminology for broadcast tree sets The difference consists mostly of keeping track of trees instead of paths in the algorithms.

4.2 Network State and Useful-state Sets

In the mind map of Fig. 4.1, we see the communication architecture in the top left corner. It consists of network topology and useful-paths set, a subset of the all-paths set, derived from the terminal arrangement, as determined by the designer. The useful-paths set is possibly diminished by the designer, for reasons of computational efficiency or good sense. Now, having defined the building blocks of a CA, we can seek to control it with minimal redundancy, hence minimal cost.

Network all-state space The state of a network follows from the states of its resources. If the resource states are independent, the *network all-state space* $S_{\mathfrak{a}}$ is the Cartesian product of resource state sets. $S_{\mathfrak{a}}(G_{NW}) = \prod_{i=1}^{|R|} S_{\mathfrak{r}}(r_i)$. ($\mathfrak{a}$ stands for *all-states*). If the switches are all of same type, $|S_{\mathfrak{a}}| = |S_{\mathfrak{r}}|^{|R|}$. In Fig. 4.2, for instance, with 4 6W3T switches and 7 states per switch, in total, $|S_{\mathfrak{a}}| = 7^4 = 2401$ states exist.

Useful state and useful-state set A CA is in a *useful state* $\mathfrak{u}$ if all paths realized by the switches belong to the useful path-set. The *useful-state set* $S_{\mathfrak{u}}(A_C)$ is the set of all its useful states. By definition, $S_{\mathfrak{u}}$ maps one-to-one to the set of concurrent path-sets S_{CPS}.

Fig. 4.8 shows that not all products of individual resource states correspond with a useful state. For instance, if switch `i4` realizes a path from `a2` to `b2`, switch `i3` is always disconnected. If switch `i3` has a path from `i1` to `i4`, switch `i4` always has a path from `i3` to `b2`. Other useless states are those where adjoining switches do not make

a path, e.g. because they are oriented in opposite. Consequently, the useful-state set maps to a *subset* of network all-state space.

Useful-state encoding The useful-state subset $S_{\mathrm{u}}(A_C)$ is often much smaller than the total network state space. When transporting state information to control the network, it is cheaper to encode it in one-to-one relationship to useful state, and not transport full network state information. We call the encoding of control *USE* when it defines only useful states.

Given the topology and useful path-set, USE is of minimal redundancy, because a state outside the useful-state set has no inter-terminal path. Therefore it should not be encoded.

In the network state corresponding to the null path-set, no resources are allocated. For power-saving, it is important that such a state can be encoded. We will always, conventionally, include it in the useful state set.

Path-set lookup table A *path-set lookup table (PSLT)* is a map from a positive integer code, the *useful-state code* to the SCPS. 0 always maps to the null path-set, by convention. We can choose the codes sequentially, or to have the lowest total entropy, or be the easiest to decode. Path-set lookup tables can be constructed by combining the tables of subnetworks that have a known PSLT or by useful-state analysis, described below.

4.3 Useful-state Analysis

In the map of Fig. 4.1, the communication architecture implies the useful-state space of the CA, which is a subset of the all-state space, but at this stage, the useful-state space is not yet fully determined.

Useful-state Analysis (USA) is the determination of the complete set of concurrent path-sets, the path-set lookup table, the transfer-set lookup table, and the switch lookup tables, from the communication architecture (CA) $A_C = (G_{NW}, S_{UP})$.

Path allocation graph A *path allocation graph (PAG)* is a simple undirected graph $G_{PA}(A_C) = (V, E)$, The vertices V are the members

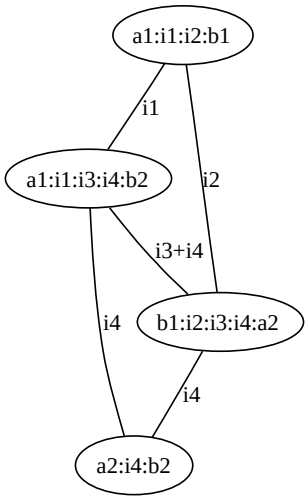


Figure 4.9: Path allocation graph for the CA of Fig. 4.2.

of the useful path-set. $V(G_{PA}) = S_{UP}(A_C)$. An edge exists between each pair of useful paths with a common resource. $E(G_{PA}) = \{\forall(p_1, p_2), p_1, p_2 \epsilon S_{UP}(A_C) \mid (\exists r \mid r \epsilon p_1 \wedge r \epsilon p_2)\}$. Fig. 4.9 shows the PAG for our example communication system with 4 terminals and 2 terminal classes.

Graph concepts In graph theory, *independent sets* are sets of vertices not incident on a common edge. *Maximal* independent sets are independent sets not contained in any other independent sets.

A *maximum* independent set, is an independent set of maximum size. A maximum independent set is a maximal independent set, but the reverse is not necessarily true [118].

The *independence number* $\alpha(G)$ is the maximum size of an independent set. We also define the *concurrency number* $\gamma(G)$ as the maximum number of concurrent paths in a graph. $\gamma(G)$ can often be determined by inspection. This is important since knowing $\gamma(G)$ limits the run time of the USA algorithms.

Concurrent path-sets in a CA by definition correspond to independent sets in the PAG. Maximal independent sets in the PAG, not contained in any other independent set, are termed *maximal path sets* in the CA. The size of independent sets in the PAG, $\alpha(G_{PA})$, is bounded by $\gamma(G_{NW})$. Since some of the maximal number of concurrent paths may

not be useful, $\alpha(G_{PA}) \leq \gamma(G_{NW})$. We call the maximal number of concurrent paths the *concurrency number of the communication architecture*, $\gamma(A_C) = \alpha(G_{PA}(A_C))$.

USA algorithm Finding the PSLT entails

1. constructing the PAG;
2. obtaining all independent sets of vertices in the PAG (which are paths in the CA);
3. (if the concurrent path-set contains undirected paths:) combining paths of both directions into new sets of directed paths;
4. numbering the sets of paths, which are concurrent path-sets, with useful-state codes.

Constructing the PAG in step 1) is a matter of enumerating the useful paths as vertices, and introducing an edge whenever pairs of useful paths have resources in common. The technical aspect of step 2) is to be described in Section 5.2. The rationale for step 3) is computational: if the concurrent path-set contains bidirectional paths, one could replace each by two directed paths and proceed as above, but without step 3). One then has up to twice as many useful paths, leading to far higher complexity, as Chapter 5 will show. It is better to keep the paths undirected at first and recombine the paths of both directions later in new sets of directed paths, with little computational burden. For instance, the undirected path-set {`a2-i4-b2, a1-i1-i2-b1`} recombines to four directed path-sets {`a2->i4->b2, a1->i1->i2->b1`}, {`b2->i4->a2, a1->i1->i2->b1`}, {`a2->i4->b2, b1->i2->i1->a1`} and {`b2->i4->a2, b1->i2->i1->a1`}.

The result of the USA algorithms is a lookup table for the set of concurrent path-sets, the path-set lookup table (PSLT). The chosen numbers can be used as instruction or control codes.

Example Using a USA algorithm, we find the complete directed PSLT for the communication architecture of Fig. 4.2. It is depicted in Fig. 4.11, and listed in Fig. 4.10. We find that Fig. 4.8 did represent the set of maximal path-sets, but not the complete set of concurrent path-sets, after all.

```
0
1 a2->i4->b2  a1->i1->i2->b1
2 b2->i4->a2  a1->i1->i2->b1
3 a2->i4->b2  b1->i2->i1->a1
4 b2->i4->a2  b1->i2->i1->a1
5 a1->i1->i2->b1
6 b1->i2->i1->a1
7 a1->i1->i3->i4->b2
8 b2->i4->i3->i1->a1
9 a2->i4->b2
10 b2->i4->a2
11 b1->i2->i3->i4->a2
12 a2->i4->i3->i2->b1
```

Figure 4.10: A listing of the complete directed path-set lookup table (PSLT).

Effect of useful-state encoding Instead of transporting 2401 states to the 4 switches, we only need to transport 13 useful states. In terms of control wires, using USE, we need only 4 wires, instead of 12.

PSLT reduction With every directed path corresponds a directed transfer. The reduced PSLT corresponds with a transfer-set lookup table (TSLT) that has a unique set of paths per set of transfers. Reducing the PSLT entails removing from the PSLT all but one concurrent path-set that covers the same concurrent transfer-set, and (optionally) removing concurrent path-sets whose correspondent concurrent transfer-set is never required by the ISA. In our example PSLT of Fig. 4.10, no reduction is required. The TSLT is listed in Fig. 4.12.

Omitting the control codes, the TSLT can be transformed to a *transfer-compatibility hypergraph (TCH)*, a hypergraph $H_{TC}(A_C) = (X, D)$ where the vertices $X = V(G_{PA})$ are the useful paths and the hyperedges D are the useful path-sets. Fig. 4.13 shows an example TCH, for the TSLT of Fig. 4.12. In this case, it is a graph, not a hypergraph, since no conflict involves more than 2 transfers. (No useful path-set counts more than 2 useful paths.) The TCH is the specification for transfer compatibility that the scheduler needs to observe; we will meet it again in Section 7.3 on design flow.

Referring back to the map of Fig. 4.1, we have after useful-state anal-

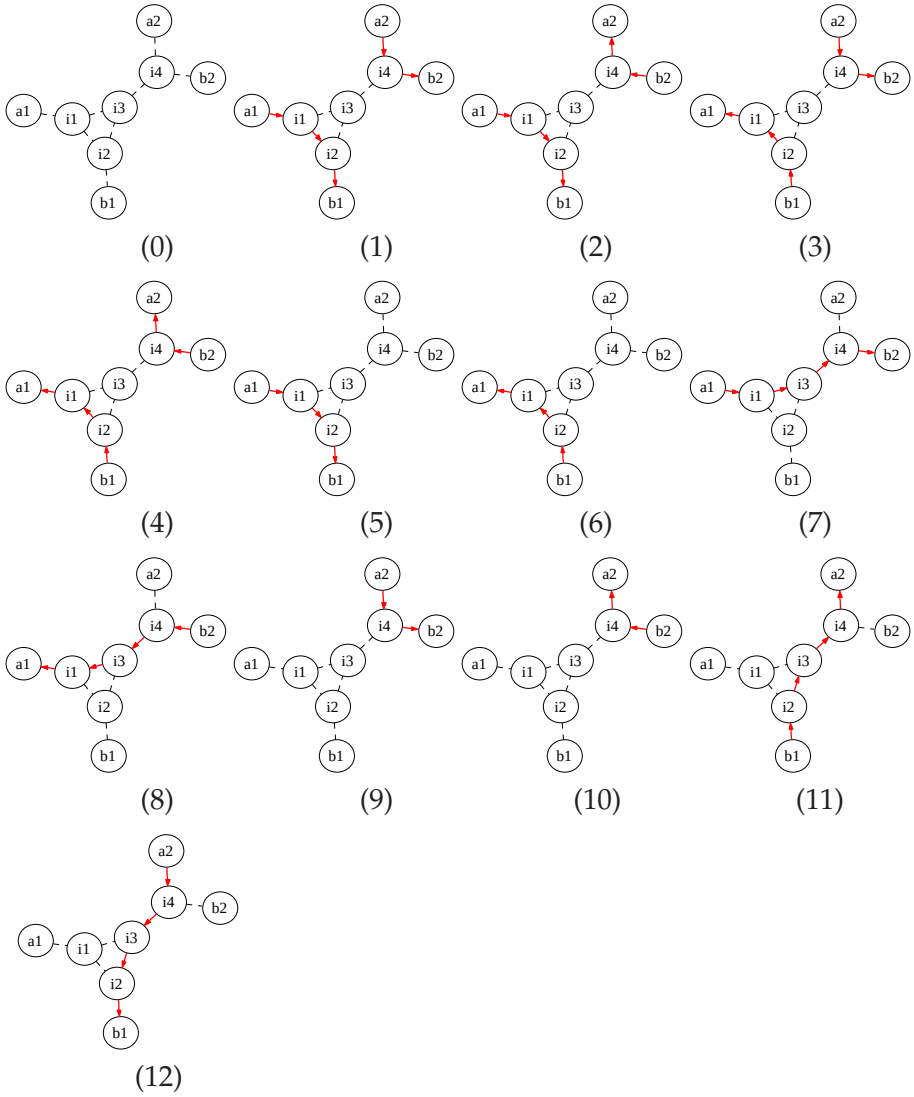


Figure 4.11: The set of concurrent path-sets (SCPS), with their useful state codes.

ysis, arrived at the top right-hand corner: the set of concurrent path-sets has been determined, and control codes have been assigned to each useful state, arriving at the path-set lookup tables. Optionally, at the discretion of the designer and in function of ISA or terminal arrangement, the set of concurrent path-sets, may be reduced and a transfer-set lookup table may be arrived at, shown in the bottom right-hand corner.

```
0
1 a2->b2  a1->b1
2 b2->a2  a1->b1
3 a2->b2  b1->a1
4 b2->a2  b1->a1
5 a1->b1
6 b1->a1
7 a1->b2
8 b2->a1
9 a2->b2
10 b2->a2
11 b1->a2
12 a2->b1
```

Figure 4.12: A listing of the transfer-set lookup table (TSLT).

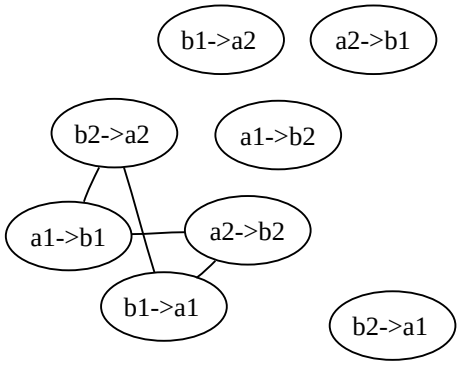


Figure 4.13: Transfer compatibility hypergraph for the TSLT of Fig. 4.12. Since the concurrency number is 2, it is also a plain graph.

4.4 Communication Architecture Bandwidth

We are now able to answer the question: when is a communication architecture fixed-bandwidth, and when is it variable-bandwidth? This can be established, by USA, from the CA: the network topology and the chosen set of useful paths, with some the concurrent path-sets removed by reduction, at the designer's discretion. The property of bandwidth variability is thus acquired by design. USA results in a canonical PSLT which describes all the useful states of the CA.

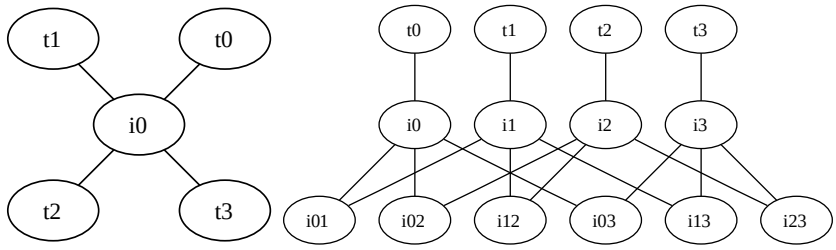


Figure 4.14: Network topologies for shared-media and point-to-point communication architectures.

t0:i0:t1	t0:i0:i01:i1:t1
t0:i0:t2	t0:i0:i02:i2:t2
t0:i0:t3	t0:i0:i03:i3:t3
t1:i0:t2	t1:i1:i12:i2:t2
t1:i0:t3	t1:i1:i13:i3:t3
t2:i0:t3	t2:i2:i23:i3:t3

Figure 4.15: Useful paths for shared-media (left) and point-to-point (right) communication architectures.

The CA is *fixed-bandwidth* if and if only *all maximal independent sets of the PAG have the same size*, which then is the *bandwidth* of the CA (in transfers per cycle). This implies that all *maximal* independent sets are *maximum* independent sets. In the other case, the CA is *variable-bandwidth*.

To make this mathematical definition consistent with our earlier statements (in Section 3.1.2), we must show that shared-media and direct ("point-to-point") CAs have a PAG with maximal independent sets of constant size. This can be derived by performing USA, which in both cases is trivial. Two topologies with four terminals are shown in Fig. 4.14: one with a common, shared resource (resource *i0*) for all communication and one with a resource per terminal pair (resources *i01-i23*), making it point-to-point. For the point-to-point network, per terminal is added an extra resource (resources *i0-i3*), to model the assignment of a channel to a terminal. Without it, the function of terminal and channel would not be properly separated and our methodology's requirement that "terminals have degree 1" violated (cfr. Section 4.1). The terminal arrangement is all-to-all; so we choose the concurrent path-sets shown in Fig. 4.15. This yields the PAGs of Fig. 4.16 and the

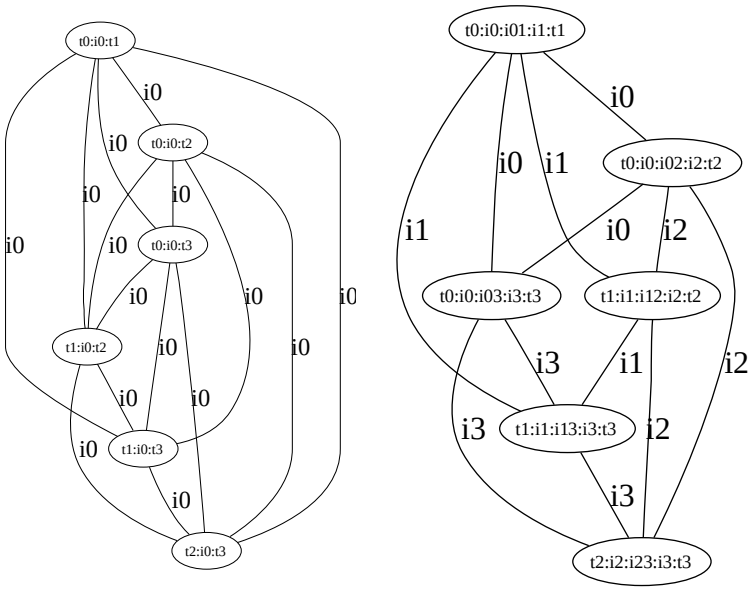


Figure 4.16: Path allocation graphs for shared-media and point-to-point communication architectures.

```
t0:i0:t1      t0:i0:i01:i1:t1 t2:i2:i23:i3:t3
t0:i0:t2      t0:i0:i02:i2:t2 t1:i1:i13:i3:t3
t0:i0:t3      t0:i0:i03:i3:t3 t1:i1:i12:i2:t2
t1:i0:t2
t1:i0:t3
t2:i0:t3
```

Figure 4.17: Maximal (and maximum) independent path-sets for the path allocation graph of shared-media (left) and point-to-point (right) communication architectures.

maximal independent sets shown in Fig. 4.17. These are all of the same size: the concurrency number of the CAs, which is 1 and 2, respectively (and for the point-to-point communication architecture not 3, since out of 3 terminals at most two terminal pairs can be chosen). The maximal independent sets are also maximum; their sizes, 1 and 2, are also the respective fixed bandwidths of the CAs in transfers per cycle.

For a CA with fixed bandwidth N, the scheduler can schedule any number from 0 to N transfers per cycle. The number of choices it has, the useful states, can be derived from the PSLT: for the shared media

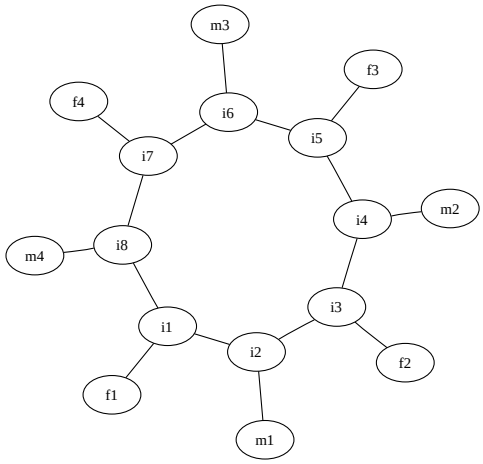


Figure 4.18: Communication architecture with circular topology.

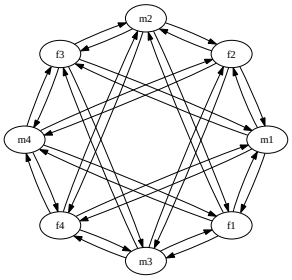


Figure 4.19: Terminal arrangement for functional units and memories.

CA, it is 7; for the point-to point, 10.

4.5 Another Example: Circular Topology with Two Terminal Classes

A circular topology has some advantages over a linear topology, at little extra cost: for the price of one extra section, one obtains the possibility to carry many more concurrent transfers. Fig. 4.18 depicts the network of a CA with two terminal classes: functional units (*f1-f4*) and memories (*m1-m4*). Each functional unit must communicate in two directions with each memory, as is indicated in the terminal arrangement of Fig. 4.19. Up to four concurrent transfers are possible, for instance the transfers `f1->m1`, `f2->m2`, `f3->m3` and `f4->m4`.

The useful-paths set S_{UP} has 32 paths. We have retained all paths from the all-paths set since even the longest paths may offer a useful detour in our terminal arrangement. The PAG for this CA has 32 vertices and 441 edges; it is shown in Fig. 4.20, if only to convey the complexity and the need for a good algorithm to find the independent sets. The *edge density*[2] is 0.43; edge densities of PAGs have been found to range

[2]Edge density $\delta = |E|/|V|^2$ is a measure of whether a graph is computationally represented most efficiently by an adjacency list ($\delta \leq 1$) or by a adjacency matrix ($\delta > 1$) [106].

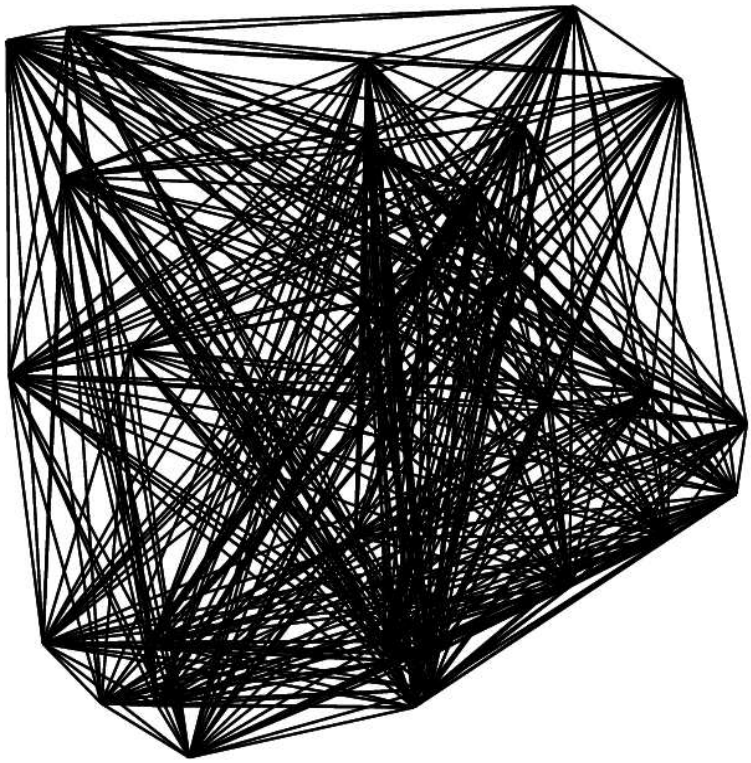


Figure 4.20: Path allocation graph for the CA of Fig. 4.18, with circular topology and a bipartite terminal arrangement.

from 0.1 to 0.5. This is smaller than 1, making the graphs low edge-density and computationally best represented by an adjacency list. We have seen that the concurrency number is 4; therefore the independent sets have at most 4 members. Using the maximum independent path-set (MIPS) algorithm (program: *mkpslt -D*), the PSLT is found in 7.8 s. USA reveals that the CA has 513 useful states. Some of the concurrent path sets are redundant; reducing the PSLT yields 478 states. Thus, $\eta_{UE} = 1 - (9/(8 \times 3)) = 62.5\%$. The CA can be driven with 9 wires instead of 24.

The bandwidth of the CA is variable between 1 and 4 transfers per cycle, as seen from the undirected maximal path-set table, which has 46 entries: 8 of size 1, 20 of size 2, 16 of size 3, and 2 of size 4. The canonical transfer-set table, reduced to include only one path-set per transfer set, has 481 entries, reduced from 513.

One wonders whether this CA could have been made fixed-bandwidth, by reducing the path-set lookup table. There are $2^4 = 16$ terminal pairs, out of which we can choose 3 pairs in $\frac{16!}{3!\times 13!} = 560$ ways. We can also choose 2 pairs in $\frac{16!}{2!\times 14!} = 120$ ways, or 1 pair in 16 ways. (Mark that some path-sets of given size are subsets of the undirected maximal path-sets of a greater size.) The sizes of the fixed-size undirected maximal path-sets are not large enough to support a fixed bandwidth larger than 1. We could have seen this from the topology: any diagonal transfer, like *f1-m3*, blocks other diagonal transfers like *f2-m4* under the prevailing terminal arrangement.

Strictly speaking, we could have obtained a single transfer per cycle version of the CA by looking only for 1-maximal independent sets. 32 undirected independent sets can be found. This can be reduced by half, since each transfer can be done using each of two alternate paths in each direction: clockwise and counterclockwise. Ultimately, this leads to a canonical PSLT with 65 entries.

4.6 Figures of Merit

The concept of useful states allows us to define figures of merit for a CA independent of the applications running on them. The figures allow us to quantify a design early in the design process.

4.6.1 Useful-encoding Efficiency

One figure of merit, Useful-encoding Efficiency (UEE) is the amount by which useful-state encoding reduces the number of control wires, relative to the number of wires before reduction. For resources of the same type, whose control states are unrelated,

$$\eta_{UE} = 1 - (\lceil \log_2 |S_{\mathfrak{u}}| \rceil / (|R|.\lceil \log_2 |S_{\mathfrak{r}}| \rceil)), \tag{4.1}$$

with $|S_{\mathfrak{u}}|$ the number of useful states, $|R|$ the number of resources and $|S_{\mathfrak{r}}|$ the number of states per resource. Assuming all resources are of the same type, the number of bits required to control $|R|$ switches, thus the width of the control plane, is

$$W_{CP} = (1 - \eta_{UE})|R|.\lceil \log_2 |S_{\mathfrak{r}}| \rceil. \tag{4.2}$$

η_{UE} ranges from 0 to 100%. In our first example, of Fig. 4.2, $\eta_{UE} = 1 - (4/(4 \times 3)) \approx 67\%$, meaning that, using USE, the resources need

only 33% of the number of wires required when USE is not used. The number of wires $W_{CP} = 4$. In our second example, of Fig. 4.18, $\eta_{UE}(8 \times 3)) = 62.5\%$.

4.6.2 Intrinsic Sectioning Gain

To define Intrinsic Sectioning Gain (ISG), we return to the concept of sectioning gain already mentioned in Chapter 3. We develop a view of sectioning gain which is independent of program, data and geometry, and thus also of power-aware placement. The *sectioning gain* G_S is the energy saved by sectioning, divided by the energy consumption as it was before sectioning. Because of Eq. (3.3), we have

$$\begin{aligned} G_S &= \frac{E_{xp_{unsect}} - E_{xp_{sect}}}{E_{xp_{unsect}}} = 1 - \frac{E_{xp_{sect}}}{E_{xp_{unsect}}} \\ &= 1 - \frac{\sum_{\forall switched-on\ sections} \alpha_s l_s}{\sum_{\forall s} \alpha_s l_s}. \end{aligned} \tag{4.3}$$

Intrinsic and program-specific sectioning gain To be independent of geometry and application, the section lengths l_s and per-section activities α_s must disappear from (4.3), making the sectioning gain inherent to the communication architecture. We call this the Intrinsic Sectioning Gain (ISG). For this, we assume *(a)* α_s to be either 0 (for sections not switched on) or 0.15, the estimated value α_i from Eq. (3.4), and *(b)* l_s to be either 0 (for terminal drop-in sections, where the resource is close to the terminal) or unit length (between resources).

$$G_{IS} = 1 - \sum_{\forall switched-on\ sections} \frac{1}{\sum_{\forall s'} 1}, \tag{4.4}$$

where s' is a section of the resource network, excluding terminal drop-in sections.

In other words, ISG is the length of unused (sectioned-off) resource network sections, normalized to the total section length of the resource network, making abstraction of geometry, and averaged in time over the duration of the application. It is the sectioning gain under *activity-* and *placement-neutral* conditions.

This average can be further simplified by making one more assumption, that of *schedule-neutral* conditions, i. e. that all useful states have

equal probability. In fixed-bandwidth CAs, this means that all transfers have equal probability, which can be true, for instance in a memory hierarchy where all memories have the same probability of access. In any case, schedule-neutrality is the best prior estimate we can make, not knowing the real distribution of useful states which depends on the compiler and the program. We define *schedule-neutral ISG* G'_{IS} as

$$G'_{IS} = \frac{1}{|S_{\mathrm{u}}|} \sum_{\forall S_{\mathrm{u}}} (1 - \sum_{\forall s'} \frac{\mu_{S_{\mathrm{u}}}(s')}{\sum_{\forall s'} 1}), \tag{4.5}$$

where $\mu_{S_{\mathrm{u}}}(s')$ is the membership function for a wire section s' to belong to the concurrent path-set corresponding to useful state S_{u}. G'_{IS} is the average length of switched-off sections, made over the set of concurrent path-sets, thus over the useful states. The real sectioning gain is (sometimes much) better than G'_{IS}, but too dependent on the program or compiler for our purpose. If power-aware placement is done properly, it should never be worse.

Power-aware placement Sectioning gains in the data plane after power-aware placement, reported in [48, 49, 57, 59], ranged from 44 to 85%. Since calculated schedule-neutral ISG was in the range of 20 to 40%, the program-specific gain $G_{PS} = G_S - G'_{IS}$ can be as high as 45% above schedule-neutral ISG.

Sub-word selection If a CA has a word-wide bus network but the ISA features transfer instructions operating on sub-words, an appreciable amount of data plane transport energy can additionally be saved by disconnecting the parallel parts of the sections for the unused parts of the sub-word. Control-wise, this is cheap, as control of sub-word selection is grouped per data word.

Because power-aware placement and sub-word selection are data plane techniques, we can, in topological analysis, consider only schedule-neutral ISG, and see it as a separate figure of merit that is proper to the CA.

In our first example of this chapter, the CA of Fig. 4.2, $G'_{IS} = 71\%$, meaning that, in schedule-neutral conditions, 71% of wire section length is unused. In the second example, the CA of Fig. 4.18, $G'_{IS} = 52\%$.

4.7 Summary

A communication architecture is described by a network graph and a useful paths-set. The latter is selected from the all-paths set. The designer selects paths he wants to use. He can keep long paths in the set and discard them later, at the risk of introducing computational complexity during USA that could have been avoided. If the network has a resource set R and an all-state space $S_{\mathfrak{a}}$, its state can equivalently be described by its useful-state set $S_{\mathfrak{u}}$ where $|S_{\mathfrak{u}}| \approx (1-\eta_{UE})|S_{\mathfrak{a}}|$. USA yields a PSLT and control codes for each of the communication architecture's useful states.

Unless the network state allows for multiple paths per transfer, the path-set lookup table can be replaced by a transfer set table, which is smaller and thus has a still larger η_{UE}. For conciseness, we will use the term *path-set specification* also in the case where the lookup is unique, and thus defines a set of transfer sets as well as a set of path-sets.

The underlying assumption for all this is that the vertices in the network are atomic, i.e. that at most one path through a vertex is allowed to exist at one time. This includes multi-way switches, multiplexers and crosspoints, but not full crossbar switches.[3] USE requires that each vertex has its own switch lookup table, correlating the global useful state code to its own state.

The purpose of reducing a PSLT can be to remove useful states because they are not required to match the terminal arrangement. This may be because the ISA is not topology-aware, or is designed for a fixed bandwidth. If the purpose is to encode only transfers and not network state, the PSLT can be represented by a single-path, or transfer lookup table.

On arrival of a useful-state code, a switch can look up its configuration by indexing the entry in its switch lookup table. A *switch lookup table* is derived from a complete canonical PSLT. The procedure is described in Appendix A, and is important for switch synthesis.

The concept of USE can be expanded to broadcasting, where path tree sets replace path-sets. Useful-state efficiency was found to be better, since broadcasting resources have more states, while the states of vertices participating in broadcasting are more strongly interrelated. The concept of broadcasting spawns yet another type of lookup table, the *path-tree set lookup table*. Its use will by now be clear.

[3] Railway switches can be handled; Internet routers, alas, not.

4.8 A Survey of Future Work on USE

The scope of USE is different, and larger than, EESC alone. At this point, we want to digress briefly from our main field of application (SoCs) and speculate where else USE and sectioning might also be applicable. Many sorts of frequently reconfigured multi-commodity transport networks exist. We want to see where else our assumptions apply, and to which extent they might be relaxed if they do not.

Frequently reconfigurable communication networks As we have seen, sectioning can be and is used widely as a cost-saving principle, wherever transport costs are proportional to the length of a link. USE applies wherever extensive networks are controlled or monitored frequently, and the volume of state information to be exchanged is appreciable. With irregular topologies, the major advantage of USE is the ease of design-time scheduling, while redundancy is minimal at run time.

Key assumptions The key assumptions for USE are *static scheduling*, *atomicity of the network vertices* and *centralized control*. The requirement for programmed control can be relaxed to program-assisted design of control, thus USE is not limited to programmed machines. A fourth assumption, that we have *circuit-orientation*, and not "store-and-forward", which would require buffering in the vertices, can possibly be relaxed by developing the proper algorithms.

Candidate application fields Overviewing some candidate application fields, like multi-commodity transportation, Internet routing, global monitoring of IPv6 network topology, NoCs, inter-tile networks in communication-aware CMP, operating-system control of chip multithreading (CMT)/CMP, FPGA interconnect fabrics, and EESC without programmed control, we see that some are ineligible (IPv6) because network vertex atomicity does not apply. We know that EESC can be used as a "style of connection" in ASIC Register Transfer Level (RTL) networks; since it can be controlled by ad hoc communication sequencers. this opens up interesting avenues in FPGAs. Elsewhere, like in NoCs and CMPs, vertex atomicity is at the designer's discretion. If we had algorithms for USA with "store and forward" mode of operation, EESC could be applied in these fields. This would allow more

power-efficient use of communication resources in NoCs and CMPs.

A number of application cases come to mind; for instance, routing in NoCs and CMPs for irregular topologies. As we have seen in Section 2.2, deterministic routing, either source routing or distributed, features a lookup table at each vertex. These tables increase in size with network size and complexity. Also, with source routing, headers greatly increase in size with large and complex networks. In NoCs with irregular topologies, significant savings might be obtained from route specification with minimal redundancy.

CMT/CMP processors with non-uniform memory access (NUMA) are of research interest today. A CMT/CMP processor consists of computing nodes with L1 caches, while L2 and lower-order caches are shared by different (or all) nodes. NUMA architectures are designed to surpass the scalability limits of the symmetric multi-processing (SMP) architecture. In a SMP, all L2 accesses use the same shared-media memory bus. NUMA alleviates this bottleneck by connecting the various computing nodes and caches by a more specialized communication architecture. This creates the paradigm of a local memory and remote memories with different levels of remoteness. Control over such a communication architecture is centralized and originates from the operating system. USE, when adapted for store-and-forward operation, could be used for discovery, description, scheduling and control of advanced and irregular NUMA topologies.

Chapter 5

Algorithms for Path-finding and USA

A computer architecture is its own punishment.

In this chapter, we describes specific algorithms developed for our design framework, and their implementation, in detail. We present algorithms for finding all routes in a network, needed to determine the all-paths set of a communication architecture, and for deriving a path-set lookup table from a path allocation graph (PAG), that is to say for Useful-state Analysis (USA).

Alternative USA algorithms will also be explored, based on graph representations other than the path allocation graph (PAG), and approaches other than graph theory.

OUR design framework for control of EESC needs to built on correct concepts, but, equally important, we must be able to perform design procedures in reasonable time. As we shall see, we are hampered by computational complexity which is essentially non-polynomial.

5.1 Path-finding Algorithms

To define a communication architecture (CA), it is necessary to determine the set of all paths between terminals. We can then choose the useful paths for the terminal arrangement at hand, and arrive at the useful-paths set. For acyclic network topologies, determining the all-paths set is trivial, while for many regular networks it can be done

analytically, but not so for the cyclic (meshed) networks that we need for implementing *survivable* architectures. Such architectures are covered exhaustively in Grover [41], albeit in the context of telecommunications, not SoCs.

Paths, or *routes* as they are often called in this context, refer to a sequence of vertices. As seen in Chapter 4, the paths must be acyclic, thus not contain the same vertex twice. They are considered distinct if the sequence of vertices is different in any way. Note that distinctness is not the same as vertex-disjointness or edge-disjointness, criteria which are also used in literature but mean only that routes have no vertex or edge in common.

On the order of $O(2^n)$ distinct routes are possible between vertex pairs in a graph with n vertices, because of the binomial sum

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n. \tag{5.1}$$

If we have S edges and use a hop limit (or maximum path length) of $H < S$, we expect a bound of $\mathcal{O}(\binom{S}{H})$. In a communication architecture, we need only to find routes between pairs of vertices that are terminals; with T terminals out of n vertices, there are $\binom{T}{2}$ of those pairs. Two algorithms have been found usable: find distinct routes (FDR) and find all routes (FAR).

Grover's find-distinct-routes algorithm Studying literature, we found that the body of algorithmic research is concerned with finding *shortest*, rather than *all* routes. Grover mentions it is theoretically possible to apply a k-shortest distinct route algorithm, which by itself is known to be efficient for the path-finding problem, repetitively, starting from an initial set of k-shortest distinct routes, and performing what is in effect a depth-first search on each edge. He warns that it is impractical to do, since the number of distinct routes explodes combinatorial as the hop count increases. He advises to use instead his own find-distinct routes algorithm, also a depth-first search. The algorithm specifies a terminal pair and then seeks to find all distinct routes, up to a certain hop limit.

The modus operandi is interesting enough to reproduce here from Grover [41]. It starts at a source vertex, and takes an excursion out on any edge incident at the source, marking the edge as having been visited. At the next edge thus visited, this is repeated. At each vertex, the rule is to take an unmarked edge, mark it, and head down further

to a next vertex if possible. Whenever the target vertex is discovered, the process records the route and backs up to the penultimate vertex on the route. If all edges are marked at that vertex, it backs up again. At the first such vertex found during this backtracking, where there is an unmarked edge, the search branches down that edge and continues as above.

Anytime the search depth reaches H hops without discovery of the target, or if a loop is discovered in the route, the process also backs up. A potential loop is discovered when the unmarked edge currently branched upon leads to a vertex that is already a parent of the vertex from which branching just occurred. In this case the edge is marked and the branch is not taken. An important detail is: as the process backs up from any vertex, all edges at the vertex that it is leaving, except the one to the parent, to which it is returning, are *un*marked upon departure from the vertex.

This is the only method found in literature to establish *all* distinct routes in a topology, apart from brute-force.[1] It has been established that the algorithm can work with even non-planar graphs; in SoC communication, topologies are anyhow expected to be planar. For small graphs, it can work well, for instance for our example on p. **??**. It can ultimately find routes with long hop counts, but also often fails to reach the end of the route in a reasonable time.

Find-all-routes algorithm Looking for alternatives, we found that, with careful programming, all routes in a sufficiently small network can be enumerated by a brute-force approach: making the combinations of all terminal pairs, permuting the internal vertices to yield all candidate routes and discarding them unless all edges in the candidate route are present in the network. This is is called the FAR algorithm.

The feasibility of FAR depends on memory-efficient generation of combinatorial sequences. With large networks, time and memory resources are likely to fail spectacularly, though. Prudence is thus advised: one should start with small hop counts and gradually increase

[1] Grover's FDR algorithm is of Hänsl-und-Gretl-type. (A continental European reader would recall "Thumblin" instead, but the English version of that fairy tale happens not to mention breadcrumbs.) It is a charming thought that such ancient prescriptions are still state-of-the-art, and similar in spirit to those given, in "The Name of the Rose"[33], by William of Baskerville to his confrater Adso, as they were contemplating, in 1327, to search the labyrinth at the Benedictine Abbey. Eco dwells on the subject of medieval path-finding for a whole chapter.

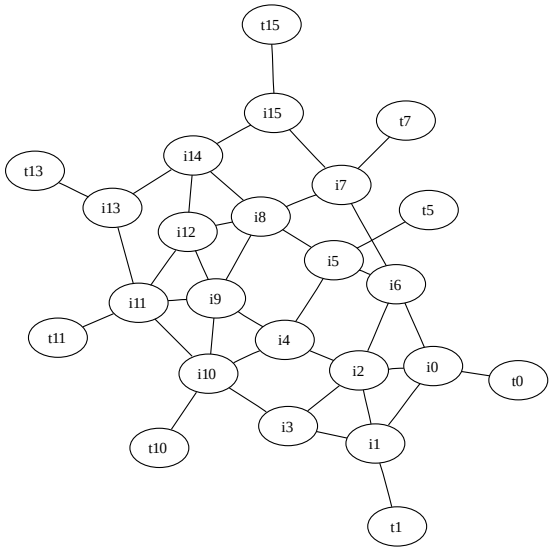


Figure 5.1: Planar graph with 8 terminals, 16 resource vertices and 36 edges between resources.

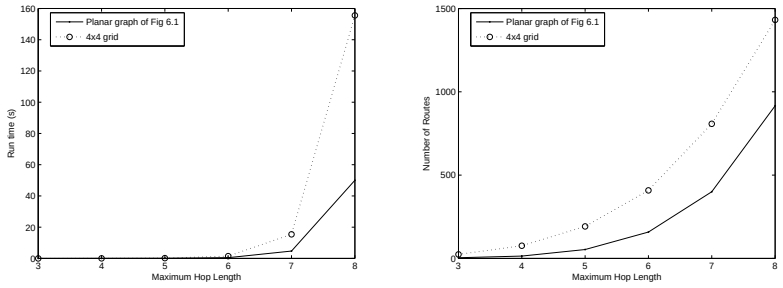


Figure 5.2: Run time required (left) and number of routes (right) for the graph of Fig. 5.1, and the 4x4 grid on p. 166, for various maximum hop counts, calculated using the find-all-routes algorithm. We see the extra run time per additional route increasing.

them.

Fig. 5.1 shows a planar graph with 8 terminals, 16 resource vertices and 36 edges between resources. In Fig. 5.2 we see the run time required[2] to find all inter-terminal routes in the graph, and the number

[2] All execution times are given for a single-threaded program on an 64-bit Intel Core Duo 2 at 1-1.67 GHz clock rate (with on-demand frequency scaling), and with a 2M L2 cache.

of routes found, for various maximum hop counts. A set of routes can easily be found provided the hop count is 7 or lower, but performance rapidly degrades with a hop count of 8 and more. This is a serious limitation, since the graph does have routes with a hop count of 16 and more, and some of those might be useful. The problem could be attacked with more sophisticated techniques, but never ultimately solved, because of the multitude of combinations of hops. Fig. 5.2 also shows results for another topology, the 4x4 grid depicted on p. 166. The extra run time per additional route increases strongly, indicating poor scalability.

If H is the hop limit, R the number of resources and T the number of terminals in a topology, then, assessing time and memory complexity of the FAR algorithm using the Standard Template Library (STL), non-strict time complexity is $O(min(H, R)2^R R!T^2)$. Memory complexity stems from the need to pre-calculate and store all terminal pairs of the communication architecture: it is $O(T^2)$.

Comparison of path-finding algorithms Both approaches are complementary: one may first finds all short routes using the find all routes, then add the longer routes using Grover's algorithm. If this has to be done, it is tedious, as it must be done for each terminal pair. Ultimately, path-finding has been done for all examples in this work. The usage forms for the tools are described in Appendix A. The job can be simplified by symmetry in the network graph. Anyway, missing routes is not very detrimental to control of the network: it only means that those routes will never be followed. If they were hard to find, chances are they are not essential to the communication architecture.

5.2 Algorithms for Useful-state Analysis

The basic way to perform USA is to determine topology and useful-path set, construct a path allocation graph (PAG) and then identify the independent sets in the PAG. These correspond with concurrent path-sets in the CA. Caution is to be exercised with undirected paths in the useful-path set (see also Section 4.3): if they are considered early on in the PAG as two paths of opposite direction, this needlessly burdens the algorithms. It is better to see undirected paths only as paths directed in opposition in the last stage of the algorithm, when undirected path-sets are recombined into sets of directed paths.

Finding in a graph a single independent vertex set, even a maximal independent set, can be solved in polynomial time by a greedy algorithm [118]. For finding *all* independent or maximal independent sets, we know no polynomial-time algorithm.

5.2.1 Maximal path-sets and PAGs: three basic methods

We will describe three variants that are non-polynomial: the independent path-set (IPS) algorithm, the maximal independent path-set (MIPS) algorithm and the path-powerset algorithm (PPS) algorithm. The second gives most insight, also in practical situations of CAs to be analyzed. All can be applied equally well on CAs with broad- or narrowcasting; the concept of a path is then replaced by a (broadcast) path-tree, that is perforce directed. The design flow and working method remains the same. Actually the concept of broadcasting resides in the PAG, not in the algorithm. (Recall that with broadcasting, the concept of a "path tree" replaces a path. Once the PAG is constructed, the allocation problem is the same for paths as for path trees.)

Before we select any of the three variants, we want to ensure that our algorithms are sufficiently optimized. We will be applying a common strategy of optimization [73], prescribing to proceed in the simplest and most straightforward way, make the algorithms *work* first, then apply it to various problems, first of small size and then larger, look at the problems of scaling, and to solve those in the order that they present themselves. We will discuss the implementation of the algorithms, illustrate the scalability problems encountered, show how those can be alleviated, and then present the time complexity of the algorithm that we will ultimately select as best.

Independent path-set algorithm (IPS) Independent sets can be found directly, by brute-force construction.The IPS algorithm is written down as pseudo-code in listing 5.1.

Maximal independent path-set algorithm (MIPS) As stated before, maximal independent sets are independent sets not contained by any other independent sets. Maximal independent sets in the PAG correspond to maximal concurrent path-sets in the CA.

The intermediary step of finding maximal independent sets could be more efficient than calculating independent sets by brute force, since

```
// find all k-maximal independent sets by brute
   force

for all sizes n of vertex sets from k downto 0
  for all vertex sets S of n-combinations
    if set S has 2 or more members
      for each 2-combination (a,b) of members
        if a and b are adjacent
          continue with next vertex set S
      record that set S is independent
    else if set S has 1 member
      record that set S is independent
    else if set S is the null set
      record that set S is independent
```

Listing 5.1: Pseudocode for the IPS Algorithm.

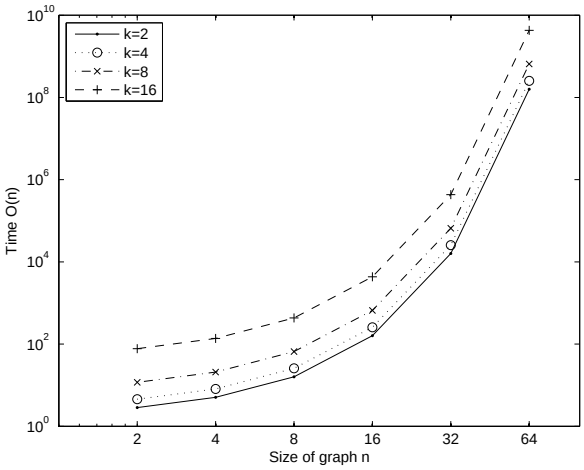


Figure 5.3: Estimated run time for Eppstein's algorithm.

most PAGs are large, which hinders any brute-force approach. The MIPS algorithm is recursive and thus easy to conceptualize. In the end, we were proved right, though counterexamples were encountered.

In MIPS we use Eppstein's algorithm [34] to find all maximal inde-

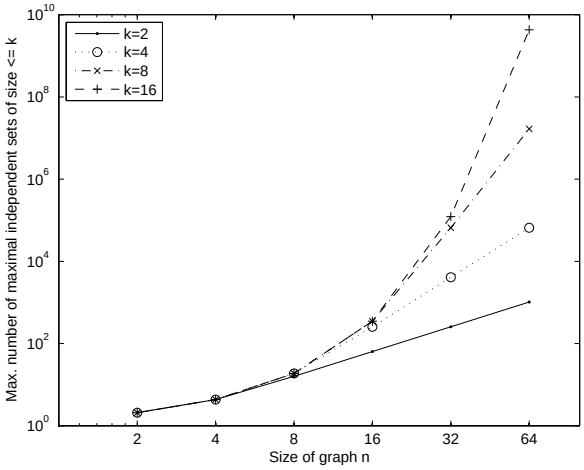


Figure 5.4: Nielsen's bound: the number of maximal independent sets of size at most k in a graph with n vertices.

pendent sets in the PAG. and runs in time

$$\mathcal{O}(n) = 3^{4k-n}4^{n-3k}. \tag{5.2}$$

This bound is tight if $n/4 \leq k \leq n/3$ and is approximately equal to $(\frac{4}{3})^k$ for $k \ll n$, although in that region the bound is not tight. The run time is shown in Fig. 5.3. In any case, the run time estimation could possibly be improved if the specific mathematical properties of a PAG, which follow from the manner of its construction, were taken into account. This has not been required up to now, and anyway exceeds the qualification of the author, who is not a mathematician, and preferred to improve the implementation, not the estimation.[3]

The number of maximal independent sets of size exactly k in any graph with n vertices is at most

$$\lfloor n/k \rfloor^{k-(n \bmod k)}(\lfloor n/k \rfloor + 1)^{n \bmod k}, \tag{5.3}$$

according to Nielsen [84], who adds that the same bound holds for the number of maximal independent sets of size *at most* k if $k \leq n/3$, and an

[3]There is an analogy here between low tire pressure and high time complexity: if you just check the pressure, you may know whether you have a problem, but if you pump the tire, you can drive.

```
boolean adjacency_map[]; // adjacency map

inline boolean function is_an_independent_set (
   bitset set, boolean* adjacency_map) {
// is the set an independent set?
    for (bitset a = set, i = 0; a != 0; a >>= 1, ++
       i)
      for (bitset b = set, j = 0; b != 0; b >>= 1,
         ++j)
        if (i > j && (a&1) && (b&1) && *(amap(j, i)
            == false))
          return false;
  return true;
}

// generate all subsets of vertices
for each subset S of the vertex set
  if (is_an_independent_set(S, &adjacency_map))
    record that set S is independent;
```

Listing 5.2: Pseudocode for the PPS Algorithm.

approximate bound of $3^{n/2}$ exists if $k > n/3$. These bounds are depicted in Fig. 5.4.

Before running the algorithm, k can be set equal to $\gamma(G_{NW})$, the concurrency number of the network used to construct the PAG, As discussed in Section 4.3, no more maximal independent paths in a set can exist for the PAG than there are concurrent paths possible in the topology. This fact reduces the expected and actual run time of Eppstein's algorithm by a large amount.

Path-powerset algorithm (PPS) Another construction algorithm to find all independent sets, PPS, more brute than IPS, can be employed. It is shown in Listing 5.2, and relies on fast execution set operations on a 64-bit integer representation of a set. The algorithm is not combinatorial at all, but just generates 2^n subsets in the fastest way (still to be determined experimentally). It checks each set for independence by shifting a copy of the set in position against the original, as shown

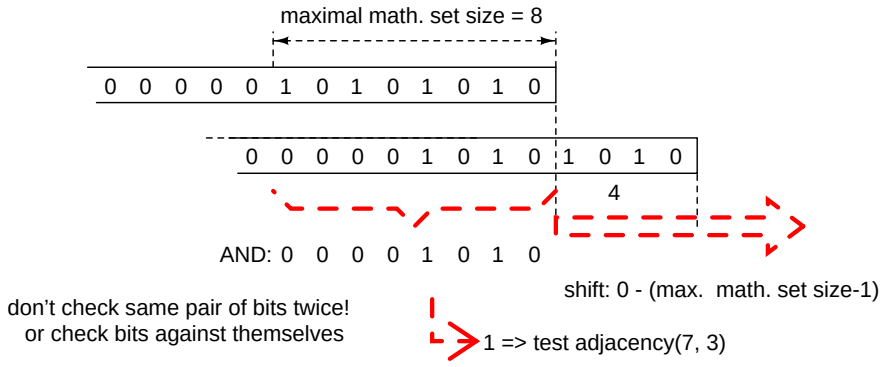


Figure 5.5: Shifting a bitset against itself to obtain all 2-combinations.

in Fig. 5.5. It consults the PAG's adjacency matrix where the positions match and the set is populated. Time complexity, for sparse graphs, is $O(2^n)$ and for fully connected graphs it is $O(2^n n^2)$; PAGs are not sparse. There is no time dependency on any maximal size k; for large k, PPS could prove to be better than MIPS or IPS. We will see below how it fares.

5.2.2 An Auxiliary Program: Dharwadker's Algorithm

Dharwadker's Algorithm and Maximum Independent Path-sets Both MIPS and IPS algorithms are not polynomial-time, and must be expected to fail at some point, for PAGs that are too large. Dharwadker [30] describes a polynomial-time algorithm to find at least one maximum independent set in graphs. He shows that every graph with n vertices and maximum vertex degree Δ must have a maximum independent set of size at least $\lceil n/(\Delta + 1) \rceil$ and that this condition is the best possible in terms of n and Δ. *Dharwadker's independent set algorithm (DIS)* [30] finds a maximum independent set in a large set of difficult graphs, and Dharwadker states the conjecture that there exists no graph for which the algorithm cannot find a maximum independent set, even though this problem has been described as NP-complete [118]. The algorithm is useful for exploration and in combination with others.

Dharwadker and the PPS algorithm PPS is used with advantage in combination with Dharwadker's IS: we spend a few seconds, before starting PPS, to find the independence number first, and then generate

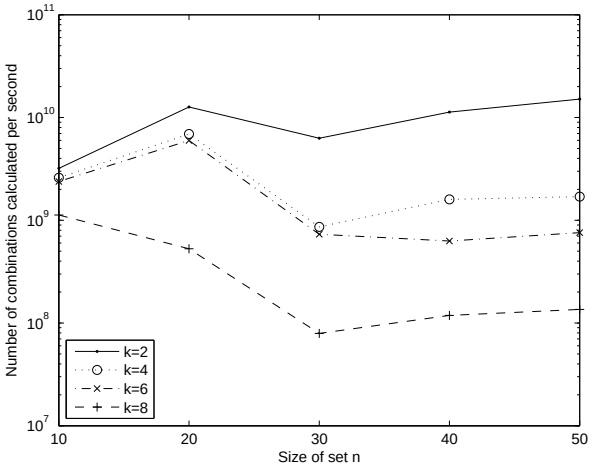


Figure 5.6: Measured run times for indexed combination algorithm.

all subsets until we have reached a size greater than the independence number.

5.2.3 Time and memory complexity

We will have to relate the time complexity of our algorithms to (3.6), the external constraint on the peak bandwidth B_{A_C} of a CA with N terminals having peak bandwidth B_T.

The bandwidth (in transfers per cycle) is exactly the concurrency number of the CA, the independence number of the PAG, and the maximum number k we need to consider in (5.3) and (5.2). External constraints on B_{A_C}, and internal constraints of the CA, imposed by graph theory, are thus related to computational complexity.

Implementation notes For our optimized implementation, we have used the STL, which uniquely offers components of known and guaranteed time complexity [107]. An O(1) implementation of combination in C++ is known [115] and has been verified: cfr. Fig. 5.6. Typically 10^9 combinations per second can be achieved with this method. It is called *indexed combination*, [4] and is central to our implementations.

[4]The rate is fairly constant over the range of n and k shown in Fig. 5.6, which is huge: between 45 and 536,878,650 combinations are possible. Thus we consider in-

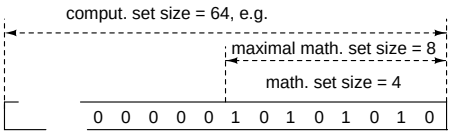


Figure 5.7: Computationally, set size and member count are different. Algorithms must be prepared to handle *M*, a maximal mathematical bit size to be expected. In this example, M must be at least 8.

Representing sets Computationally, sets can be represented by one of many types of container, like *bitsets*, *ordered* and *unordered maps* and *hashes*, *vectors*, specialized *vectors of boolean*, *lists* and a container appropriately named *set*. Which to choose depends on the access methods that will be operating on the set, the number of sets involved and the criticality of each operation for overall efficiency.

In our algorithms, immediate storage and time efficiency requirements dictate using a bitset in one form or another: either the STL own's *bitset*, Boost's *dynamic bitset* or Atlas' [8] *wide bitset*. For storage efficiency, these use only one or two machine words (on a 64-bit computer, 64 or 128 bits) to represent a set. For speed, bitsets provide fast basic set operations, including subset operations, in machine code. Our algorithms need only those, but must handle them fast and in constant time. Our algorithms also have to process many sets (as many as 8M sets with up to 16 members, in one use case), requiring efficient use of memory. From the above, we decide that no other set container than a bitset could be used and still achieve the performance attained.

With a bitset, computationally, a difference must be made between set size and member (i.e. bit) count. The *computational set size* is the width of the container (128, 64, 32 or smaller) while the *computational bit count* is the mathematical set size, i.e. the number of members. This can be observed in Fig. 5.7.

We will denote computational size of a set *S* as *S.size()*. and computational bit count as *S.count()*. The maximal bit count our algorithms expect and must reserve space for, is usually not *S.size()*, but some other figure *M* that may have to be computed from the sets themselves, in a fairly expensive for-loop. In Fig. 5.7, for instance, *S.size() = 64*, *S.count() = 4*, and *M = 8* bits need to be reserved to hold the set *S*.

Computational set size is determined statically at compile time (STL

dexed combination O(1), even if there is still 100:1 variation in run time.

and Atlas bitsets) or dynamically at run time (Boost dynamic bitset). Since the sizes of two sets operated upon must anyway be equal, the dynamic bitsets have no advantage for our purpose. Our algorithms are all limited by the implementation- and machine-dependent maximum size of the bitset, determining the number of vertices that can be handled in a PAG. With Atlas' wide bitsets, on a 64-bit computer, the limit is 128 vertices. Fokko du Cloux, the author of [8]. mentions that wider bitsets are possible only with less than optimal speed. Since 64-vertex PAGs are commonplace, using the 128-bit Atlas bitset is mandatory, if the algorithm really can withstand more than 64 vertices. PAGs with more than 128 vertices have also recently been encountered. To handle them, we will need a whole new computational approach, to be covered in future work.

Enumerating subsets Enumerating subsets is critical for all our algorithms. The number of subsets of a set S, including itself and the null set is $2^{|S|}$, the binomial sum (5.1). Here $|S|$ is the mathematical size of the set, the bit count *S.count()*.

Subsets of a set can be enumerated in at least four straightforward ways: by means of a *powerset iterator (PIT)* (also known as Gray code iterator), as a *sequence of combinations (SOC)*, by a *Samuel Beckett iterator (BIT)* (also known as binary reflected Gray code iterator), or by a *breadth-first search iterator (BFSIT)*.

PIT iteration simply involves counting in binary and looking at the binary representation. The iteration code for SOC, is shown in listing 5.3. It uses indexed r-combination of n members: the bits to be combined are copied to an n-sequence vector, of size n. The n-sequence is the larger sequence from which an r-sequence of size r is picked. The r-sequence vector initially contains sequential indices into the n-sequence; during operation, it is modified by the combination iterator. The iterator proceeds until $\frac{n!}{r!(n-r)!}$ combinations have been found. BIT is named after Samuel Beckett because the iterator script reads like the stage directions for one of his plays: starting from an empty stage, N characters from an "actor troupe" enter and exit one at a time, in such way that each subset of actors appears exactly once. The iteration code is shown in listing 5.4. The code for BFSIT is too long to reproduce; it is essentially a rephrasing of breadth-first search on graphs.

```
bitset<64> S(s);  // s is the set to combine
vector < unsigned > nseq; // n-sequence
for (unsigned i = 0; i < s.size(); i++)
  if (S[i])
    nseq.push_back(i); // populate n-sequence
const unsigned sz = nseq.size(); // math. size

for (unsigned k = sz; k > 0; --k) {
  indexed_combination it(sz, k);

  vector < unsigned > rseq; // r-sequence
  for (unsigned i = 0; i < k; i++)
    rseq.push_back(i); // populate r-sequence

  do { bitset<64> out(0UL);
    for (unsigned l = 0; l < k; l++)
      out.set(nseq[rseq[l]], 1);
    cout << out << endl;
  } while (it.next_combination(rseq));
}
```

Listing 5.3: SOC subset iterator. The set container is a `bitset`$< 64 >$.

Table 5.1: Comparison of run times to enumerate all subsets of a bitset with 16 members: C++ `or C` code, modified for $O(2^{(S.count())})$ behavior if necessary, and optimized. Standard output was piped into `/dev/null` in order to neutralize the varying effects of buffered output. All test sets are the same, except for the PIT iterator, which cannot generate all subsets of `0xaaaaaaaa` in reasonable time.

iterator	test set	static size	max. set size	math. size i.e. bit count	run time
PIT	`0xaaaaaaaa`	64	32	16	8.81 s
SOC	`0xaaaaaaaa`	64	32	16	0.093 s
BIT	`0xaaaaaaaa`	64	32	16	0.141 s
BFSIT	`0xaaaaaaaa`	128	32	16	0.386 s

```
void moves(bitset<32>& S, int n, bool enter) {
  if (n == 0) return;
  moves(S, n - 1, true); // enter
  if (enter) S.set(n - 1, 1);
  else       S.set(n - 1, 0);
  cout << S << endl;
  moves(S, n - 1, false); // leave
}

int main(int ac, char *argv[]) {
  const unsigned N = atoi(argv[1]);
  bitset<32> S(0);
  moves(S, N, true);
  return 0;
}
```

Listing 5.4: Beckett subset iterator, on `bitset`$< 32 >$.

Selecting a subset enumerator The *order of enumeration* of the sets is important and typical for each: for PIT and BIT: it is *in numerical order*, the binary code; for SOC, enumeration is from *small to large sets* or the reverse; and for BFSIT: from small to large. In fact, the order can always be reversed, by just flipping the bits. PIT is too slow. SOC, using indexed combination, runs better than $O(2^{S.count()})$. BIT runs in $O(2^{S.size()})$. BFSIT runs worse than $O(2^{S.size()})$. Running in $O(2^{S.size()})$, instead of $O(2^{S.count()})$, is catastrophic for sets S of given size that are sparse in any way, i.e. contain 0-bits anywhere. This means: for most interesting sets. Fortunately, BIT and BFSIT have indexed variations where run times depend on $S.count()$ instead of on $S.size()$, with little overhead. This is done by providing a constant-time index lookup table before iteration starts. With these modifications taken in account, we compare the four modes of iteration. SOC is remarkably efficient, thanks to constant-time combination. BIT is elegant, and has little overhead. It is recursive, but uses a predetermined amount of space on the stack. BFSIT is also recursive but has more overhead than BIT. Its useful property is that of iterating in order of size. Table 5.1 compares run times of four methods to enumerate bitsets. SOC, BIT and BFSIT all perform well; BFSIT is reasonably fast, but allocates an unordered hash map that can exhaust memory, which, as we will see, bothers us with

Table 5.2: Comparison of run times for MIPS, IPS and PPS (0ptimized C^{++} code using STL).

comm. arch.	vertices in PAG	k	δ	MIPS	IPS	PPS & DIS
circ. topol. w. 2 term. cl.	32	4	0.429	0.64 s	0.06 s	0.52 s
K-ring & all-to-all TA	61	4	0.349	0.19 s	0.18 s	40-180 s swaps
3x3 grid & all-to-all TA	70	4	0.393	0.28s	0.31 s	N.A.
FU chaining, reduc. PAG	40	8	0.097	4.96 s	0.11 s	N.A.
reduc. PAG	40	12	0.097	8.29 s	16.8 s	N.A.
reduc. PAG	40	16	0.097	8.65s s	$>$ 1h	$>$ 6h
FU chaining, compl. PAG	72	16	0.124	15077 s = 4h10m	$>$ 8h	N.A.

big sets.

Enumerating subsets in order of set size, implies that BFSIT can be broken off at a given set size. SOC has the same property which becomes even more advantageous when a series of combinations can be broken off if certain sets prove not to be independent. This is the basis of the relative success of IPS.

Profiling and inlining Profiling, subsequent inlining of code, and optimizing data structures are never fully completed. For instance, at this moment, the BFSIT used in MIPS could improve from using a better hash function, maybe a perfect hash function [18]; this is a matter of future study. We describe here our results for three algorithms, not only the best, since it cannot be excluded that an algorithm which is suboptimal in one combination of circumstances later becomes best.

Results Table 5.2 compares run times between the MIPS algorithm, brute-force construction of the independent sets (IPS) and the still more brute-force (PPS) algorithm. The PPS algorithm in the table breaks off when the size of the maximum independent path-set, predicted by Dharwadker's algorithm, is reached. For subset iterator, the version of MIPS discussed in the table uses BIT, the IPS uses SOC (which is much improved by breaking off further combinations when dependent sets are encountered), and PPS with DIS uses BFSIT. The programs have been optimized intensively. Among the test cases, the CA with circular topology and 2 terminal classes was presented in Section 4.5. The K-ring and 3x3 grid CA will be described in Chapter 8. The functional unit (FU)-chaining CA will be described in Section 8.2.1 on p. 151. The network topology is shown in Fig. 8.20. The PAG is too large to be visualized. Some algorithms are too slow for the full FU-chaining PAG to complete in reasonable time, so we also ran them on a reduced version of this PAG with only 40, instead of 72, vertices. PPS for the K-ring runs in a variable time because it runs out of memory and starts trashing memory, which depends on circumstances. In the table, 'N.A.' means the program cannot be run at all: either because the PAG contains more than 64 vertices, or because it always iterates until the independence number is reached (PPS & DIS).

The MIPS algorithm now performs best in overall and can perform USA for the difficult use case of Section 8.2.1 in 4 hours. IPS outperforms MIPS on some smaller PAGs, but fails on large ones like the full functional-unit chaining PAG. Henceforward, we will as a candidate only consider MIPS for further optimization.

Time complexity of MIPS MIPS runs in three phases: *(i)* the calculation of small independent sets using Eppsteins's algorithm; *(ii)* the calculation of all independent sets from the small independent sets, using a BIT to generate subsets of decreasing size; *(iii)* it provides the caller of the algorithm with a list iterator on the set of independent sets. The caller can copy or use the list at its own discretion. This is by itself an optimization, since copying large lists can impose a serious overhead.

Phase *(ii)* also is an optimization: the independent sets are obtained from an intermediate list of *small independent sets*, not from the list of maximal independent sets which Eppstein's algorithm can also generate. The small independent sets can possibly include sets that are not maximal; tracing reveals that this rarely happens. Using the list of small

independent sets, we avoid an extra phase of verifying their maximality.

Observing the behavior of MIPS, we see that phase *(ii)* now outweighs other phases in time complexity. Consequently, we can apply Nielsen's bound (5.3), the best we know, to the whole of the MIPS algorithm. In (5.3), we must use $n = |V(PAG)|$ and k, the independence number of the PAG, as can be found from DIS. We not yet encountered $k > n/3$ in our CAs and will only consider $k \leq n/3$. Recall that Nielsen's bound also holds for the number of maximal independent sets of size *at most* k.

According to MIPS procedure, we find the time complexity as

$$O(n,k) < \sum_{S_{CPS}(A_C)} 2^{\alpha(G_{PA}(A_C))}. \tag{5.4}$$

where $S_{CPS}(A_C)$ is the set of concurrent path-sets and $k = \alpha(G_{PA}(A_C))$ is the concurrency number of the communication architecture A_C. In the form given above, this is not so exciting, since the set of concurrent path-sets is what we are looking for. Plugging Nielsen's bound into (5.4) yields

$$O(n,k) < \lfloor n/k \rfloor^{k-n \bmod k)} (\lfloor n/k \rfloor + 1)^{n \bmod k} 2^k. \tag{5.5}$$

This is a function of $n = |V(PAG)|$ only, since k is determined as the PAGs' independence number, but we cannot determine k analytically. We only know that in our use cases $k \leq n/3$ and that $k \leq \frac{N!}{(N-2)!2!}$, the external bound, with N the number of terminals. The latter bound is rather loose.

Time complexity of MIPS is shown in Fig. 5.8. For fixed-bandwidth CAs, Eq. (5.5) is tighter than for CAs with variable-bandwidth, since (5.4) is then exact, not a bound. Reducing the time complexity of listing the independent sets from the small maximal independent set found by Eppstein's algorithm now has high priority in our future work. Memory complexity is not yet an issue with MIPS.

Multi-core multithreading Of our four iterators, SOC and PIT can be multithreaded by partitioning the iterator space; the parallelization of BIT, seen from the viewpoint of a member of Beckett's actor troupe, is similar to the dining philospher's problem and could possibly be approached that way; BFSIT cannot readily be parallelized, but the Eppstein algorithm as a whole can. Future work could modify MIPS to

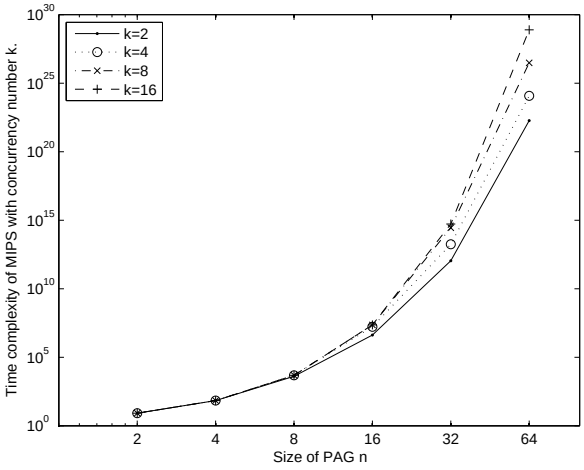


Figure 5.8: Time complexity of MIPS for a PAG with n vertices and independence number k.

run several instances of the Eppstein algorithm at once, and afterwards recombine them.

5.3 Exploration of alternative algorithms for USA

Other allocation graphs Apart from the PAG shown in Fig. 4.9, other graphs can express the relationships between resources and useful paths in a CA. The *resource allocation hypergraph (RAH)*, seen in Fig. 5.9, is the dual of the PAG: $H_{RA}(A_C) = (G_{PA}(A_C))^* = (X, D) = (E(G_{PA}), V(G_{PA}))$. Its vertices X are resources, and the hyperedges D are useful paths. A vertex is incident to a hyperedge in $H_{RA}(A_C)$ if the resource is part of the path.

Another representation of the same information is the *path-resource allocation graph (PRAG)* shown in Fig. 5.10. It is the bipartite (König) representation of the RAH. The PRAG is a bipartite graph $G_{PRA}(A_C) = (V(G_{PA}) \cup E(G_{PA}), Z)$. Its vertices are the vertices and edges of H_{RA}, and a vertex v is adjacent to a vertex e if and only if in H_{RA} the vertex v is incident to the edge e. Looking at Fig. 5.10, we see that it is a good expression of the allocation problem: if we consider the paths as 'jobs' and the switches as 'resources', the existence of an edge represents the allocation of a resource to a job.

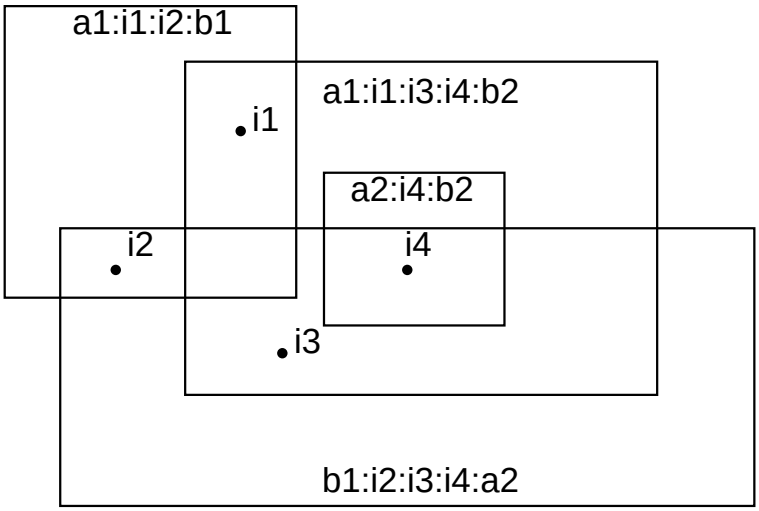


Figure 5.9: Resource allocation hypergraph for the CA of Fig. 4.2.

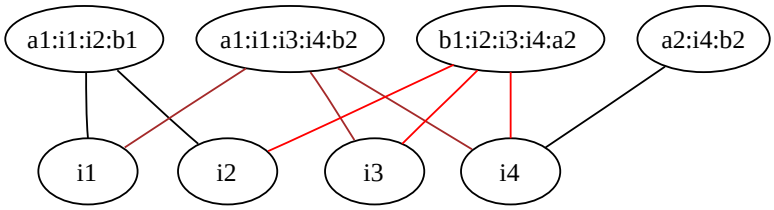


Figure 5.10: Path-resource allocation graph for the CA of Fig. 4.2.

USA on PAGs involves independent sets of vertices with no edge in common. On RAHs it involves the dual problem, the matching of sets of edges without common vertices. Fewer algorithms exist for hypergraphs than for plain graphs: the RAH approach is not expected to deliver advantages in efficiency. The advantage of a RAH representation is that it can be mixed in with other constraints on resource usage, for instance when a buffered resource is sequentially allocated, over different time slots, to a path. Voloshin [114] shows how the RAH and PRAG representations can be extended to represent resource allocation over subsequent time slots. This involves the theory and vertex-coloring of *mixed* hypergraphs.

Coloring algorithms for USA The PRAG representation has been used experimentally for another approach to USA that colored edges

in the PRAG: a correct resource allocation has edge colors which are *different* at the side of the resources, and the *same* at the side of the paths. One color,for instance the black edge color in Fig. 5.10, then represents a valid allocation for all resources. In the example, resources `i1` and `i2` are allocated to path `a1:i1:i2:b2`, resources `i3` is disconnected, and resource `i4` is allocated to path `a2:i4:b2`. Each proper coloring yields a number of useful states, and also the allocation of each individual resource in the state. A distributed edge-coloring algorithm from Grable and Panconesi cite [88] was at one point adapted for this type of edge-coloring. USA based on this algorithm has the disadvantage of not yielding all useful states at once: it must be repeatedly applied until no more new useful states are found. On the other hand, the method is distributed and suitable for parallelization with many threads.

Linear and Integer Programming Approaches using integer (linear) programming (ILP), mixed integer programming (MIP) and 1/0 MIP are advocated in [41] as yielding significant and practical tools for analysis and design of mesh-based survivable networks. Modern solution engines can handle very large optimization problems of these types, giving optimal or high quality solutions in reasonable time. However the linear and integer programming approach is not intuitive and is known to give little insight in the optimization problem at hand. This is the reason why we did not use it.

Linear and integer programming methods have arisen from the field of mathematical programming methods in the operational research community.[5] Mathematical programming methods are: first, linear programming (LP), where the objective and the constraints in the program are all linear functions, and the decision variables may take on continuous real values. Second, ILP, where one or more decision variables are restricted to integer values only. All of the objective and constraint functions remain linear. In a pure ILP all variables are strictly integer. Thirdly, in MIP, some variables are continuous and others are restricted to whole numbers or integers. A more restricted form of MIP is called a 1/0 MIP problem, where some integer variables are further restricted to take on only binary values, typically representing yes/no outcomes.

[5]Operational research is concerned with creating minimal cost (or maximal benefit) schedules or plans; such plans are called "programs". The terminology dates from the 1930s, when computer programming did not yet exist. The term "programming" in this context has little to do with its usual meaning today.

5.4 Conclusions and Future Work

When facing combinatorial complexity, one cannot win, only put up a spirited defense. Since the conception of USA, tools based on the algorithms have matured: the performance of implementations have improved, scaling down in run time, over a range of several decades. Comparing the run times of our first research prototypes and present state of the art, they decreased, for some use-cases, 70 to 600-fold for path-finding and 2000-fold for USA. This was achieved by paying attention to detail in implementation.

For USA, many avenues of advance are still open: creative use of allocation graphs, mixed hypergraphs to express time-dependent constraints on allocation, coloring algorithms, linear and integer programming, and multithreading implementations.

For path finding, the author is under the impression of presently being in a cul-de-sac, where more assistance from mathematics is needed. The FAR algorithm works for a limited number of terminals, and for use cases in this work, but scales badly. The FDR algorithm is helpful when FAR fails, but not as reliable as FAR and it sometimes needs human supervision. This will hinder USA for large irregular topologies, since analytic path-finding is possible for regular networks.

Chapter 6

Design Pattern for an Optimal Control Plane

Rejoice, rejoice, we have no choice.

CSN&Y, *Carry On*.

In this chapter, we propose our conceptual framework for EESC control plane design, guided by the paradigm of the communication processor (CP). We consider the issues that it exposes, identify the stages that must be part of the control plane, and fit them into a design pattern.

We will quantify the losses of each stage, based on our model for on-chip communication, and find some further optimizations that may become important at the far end of scalability, with many terminals, or for SoCs survivable in the face of interconnect reliability degradation.

REFERRING back to the CP paradigm and control plane model first presented in Fig. 3.9, we now need a framework for the design of small control planes. The designer's task is to convey scheduling decisions to each switch, avoiding energy losses and wire congestion. We presume the communication architecture to be configured frequently, up to once per transfer cycle.[1] If the volume of the reconfiguration information were small in comparison to the data transport information,

[1] Infrequently reconfigured communication architectures allow some obvious optimizations, like differential control of network state, that we will not explore.

then reducing energy loss during distribution would be less urgent, but it would still be important to convey the *correct* information at all times.

6.1 Design Choices

The paradigm of the communication processor allows us to settle our minds about some fundamental issues in the control plane, involving *(a)* the consequences of explicit path specification, *(b)* the timing inside the processor, and *(c)* the presence of loop buffers in the control plane. These issues concern optimization, or even simply feasibility of what we want to achieve.

Implicit and explicit path-set specification In many ISAs, each transfer-set specification is *implicitly* encoded within the transfer and operational instructions. A transfer operation, for instance for a CA based on multiple buses, will include the bus network-ID specification, the specification of the pairs of terminal sets for transfers, and the direction of the transfers. In an operational instruction on two operands, the two input transfers from a register file to the inputs of the functional unit, plus the output transfer from functional unit to register file are implicitly specified. These specifications are encoded in some efficient way, proper to the ISA.

Alternatively, it is possible that the ISA includes elements of instructions that set the control codes and the state of the CA directly. We call this *explicit path specification*. It means that the communication instruction code immediately becomes control code. This would normally be using the useful-state code, because it has least redundancy, as we saw in Chapter 4.

Implicit Path Specification Fig. 6.1 shows the form our design pattern takes with implicit path specification. Unless the CA is topology-aware, the width of the control plane at the point of fetching is zero: $W_{CP,FETCH} = 0$. With a topology-aware ISA, $W_{CP,FETCH}$ must still be small, since it only needs to differentiate between paths covering the same transfer. If the CA is composite, the control path is split before the transcoder (also the path decoder). The path decoder is integrated with other instruction decoding, including the Terminal Select Unit (TSU), and produces a single control code for all switches. Its out-

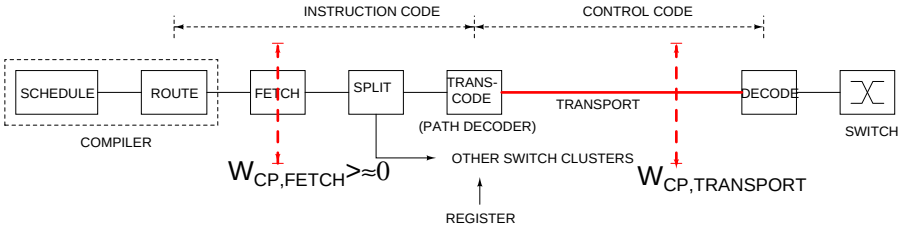


Figure 6.1: Design pattern for a control plane with implicit path specification. The width of the control path at the point of fetching is small. There is no transport loop buffering since the overhead of memory access would be prohibitive.

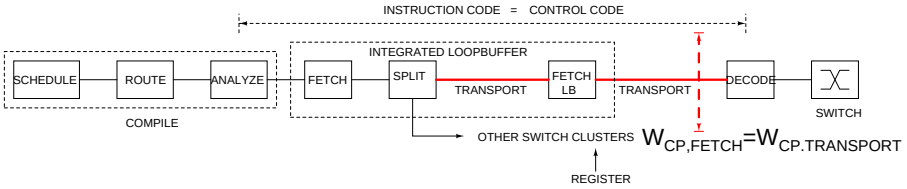


Figure 6.2: Design pattern for a control plane with explicit path specification. The width of the control path for fetching and transporting is the same. Transport loop buffering is integrated with fetch loop buffering.

put width, $W_{CP,TRANSPORT}$, is minimal due to USE. $W_{CP,FETCH} \neq W_{CP,TRANSPORT}$. The switches' decoders are synthesized individually, from information present in the PSLT.

Explicit Path Specification Fig. 6.2 shows the form of our design pattern with explicit path specification. The path specification is fetched together with other instructions. If the processor features clustered fetch loop buffers, these are integrated with the decoding of branching instructions. Clustering is organized per component CA. There is no transcoder, thus $W_{CP,FETCH} = W_{CP,TRANSPORT}$. Transport and decoding is as above.

Redundancy of explicit path-set specification One of the problems with explicit path specification is redundancy: all information is already implicitly present in the decoded communication instruction. It is an unnecessary cost to explicitly fetch control code, since a decoder could obtain the same information without extra fetching. The amount of extra information to be fetched for the benefit of the control plane, in

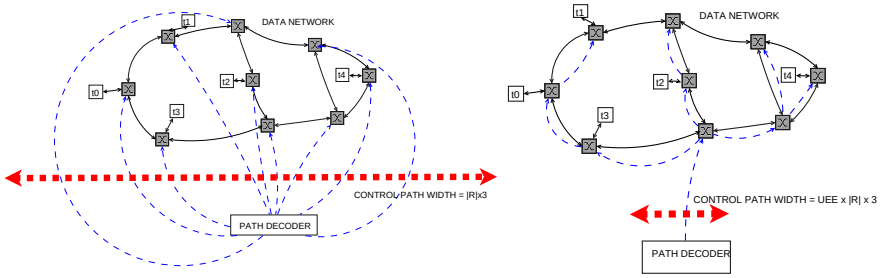


Figure 6.3: Control plane without (left) and with useful-state encoding (right): compare the level of wire congestion, the degree of difficulty for timing closure and the risk of glitches occurring. Control codes are all different without USE, all the same with USE. Also mark the reduction of the control path width.

comparison to a processor without it, is essentially zero, unless the ISA is topology-aware. In that case, the extra information amounts to the 2-logarithm of the concurrency number $\gamma(A_C)$ of the CA, at most.

Register-indirect (RI) addressing Another feasibility problem arises whenever addresses (of memory or registers) are required to select terminals in the CA. The reason is the prevalence of RI addressing mode when emitting instructions.[2] RI addressing precludes explicit path specification. If all control codes are not known at compile time, explicit paths cannot be specified.

With RT addressing and explicit path specification, then, an extra analysis stage to determine addresses and so reference data is necessary for the compiler. To develop such a compiler is an appreciable effort. Implicit path specification, on the other hand, is performed by an ordinary compiler: no development is needed.

Timing and glitches In staged pipelined processors, instruction codes are registered after fetching. A time slot of 16 fan-out 4 (FO4) delays [1] is typically available for decoding and also for subsequent transfer stages. This decoding time slot is equally available for the path transcoder. The timing of memory access stages, is too dependent on specific processor properties and CA topology to fit in an abstract treatment of the control plane. We studied it in a use case (in Section 8.1.2).

[2]Most processors, apart from stream processors, have some form of RI addressing, since it is efficient for both compiler and hardware implementation [86].

Similarly, the problem of glitches within the control plane, with data and control traveling possibly in opposite direction along the meshes of the network, was not studied. Employing useful-state code, which is identical for each switch and travels on a limited number of wires alongside the data path, helps in reducing the risk of glitches occurring and also in obtaining timing closure, as can be glanced from Fig. 6.3.

Transport loop buffering and decoding Fig. 6.2 and the principle of staged pipelining show why fetch and transport loop buffering must of necessity be separate. After reading the buffers, the decoded instruction information must be registered to be in sync with the processor's pipeline. Also, a fetch loop buffer's output must interact with the instruction branching circuitry. Thus, if transport loop buffering is to be deployed, it must use its own loop buffers. The cost of accessing these transport loop buffers must then be offset by a reduction in costs of transporting control information. Our publication [59] showed that this is not the case, given the wire-lengths and technology nodes involved.

Conclusion Use cases (in Section 8.1.3 and 8.2.1 show that explicit path scheduling leads to a sub-optimal control plane. For this and other reasons, implicit path specification is always advisable, with any conventional compiler. As mentioned before, if the tile is non-programmed (and thus not a SoC), it is possible to build a "communication sequencer" just for the purpose of controlling EESC. In that case, there is normally only direct addressing, no need for a compiler, and not a conventional pipelined timing model. Explicit path specification may then make sense.

Fig. 6.3 shows another advantage of USE, maybe not yet enough stressed before: a control plane designed with USE suffers less from wire congestion, which translates to savings on chip area.

6.2 Operation of the Control Plane

With the tasks set for the communication processor, the design space for complex communication topologies explored, and some issues of feasibility settled, we can propose a design pattern for an optimal EESC

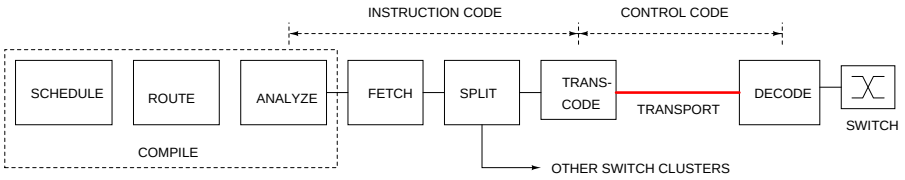


Figure 6.4: Six stages in control, including five physical stages. The role of compiler and transcoder can vary according to the type of path specification. Splitting the control plane in components CAs, i.e. clusters of switches, makes USA, and analysis in general, easier.

control plane. We first identify the stages required to design and operate a management plane, shown in Fig. 6.4.

6.2.1 Stages of Control

Six stages are identified in control plane sequencing: *compiling*, *fetching*, *splitting*, *transcoding*, *transporting*, and finally, *decoding* in the switches. All, except the first, occur in the processor at program run time. They consume chip area, latency and energy. In any attempt to achieve optimized control, these hardware costs need to be traded off between each other. The cost of development, including USA, must be set off against all hardware costs.

Compilation Tasks for the compiler include scheduling, routing, and possibly control-code prediction, i.e. analysis of the control bits to be emitted. Control-code prediction is difficult, often impossible, but only required for explicit path specification. If the CA is fixed-bandwidth, most compilers already perform scheduling correctly. Variable-bandwidth scheduling is a novelty. If it needs to be done, the compiler needs to know the TSLT.[3] Topology-aware scheduling needs of course to fit in with all other scheduling to be performed by the compiler: basic code blocks, registers, functional units, etc. The subject matter is vast and out of topic for our work.

Fetching produces instruction code for the processor. With implicit path-set specification, little or no extra costs from fetching are to be attributed to the control plane. With explicit path-set specification, we

[3]Unless the ISA is topology-aware, a full PSLT is not required at compile time.

would still have to fetch the implicit information to operate the date plane; be it redundant, it is still required.

Splitting This stage can exploit the opportunity of clustering the switches and reduce subsequent costs in transport and decoding. It has no cost by itself. A clustered control plane is recommendable if the data network consists of similar subnetworks, because of design-time efficiency of computation, but also, as use cases have shown, since it reduces decoding loss.

Transcoding produces switch control codes. The resources to be used and paths to be switched can be determined by the path decoder from implicit information and the TSU. On the other hand, if the path-set specification is explicit, we may need no transcoding.

Transporting Feeding the control code into the wires has it own costs in energy, latency and area (of the drivers).

With explicit path-set specification, fetch loop buffering can be integral to both the fetching and part of the transport stage, if there is no transcoder or splitting in-between. It can never be effective on the last stretch to the switches, since loop buffered instruction must be decoded at some point. If we still want temporal re-use after that point, we would need transport loop buffers.

Decoding In our experiments, decoding costs were found to be negligible in respect to other costs. Decoders were little more complex than the default decoder of Fig. 3.2.

6.2.2 Losses from Control

To each of the stages mentioned above is associated its own type of losses.

Fetch losses Fetch losses in area and energy are both proportional to control path width at the point of instruction code fetching. Latency is not increased by fetching, since communication instructions are fetched in parallel with other parts of instructions. In a design, fetching uses up a portion of the instruction width budget allocated by the ISA designer,

and this budget must be respected. The requirement at the point of fetching is that the width of the control plane $W_{CP,FETCH} < K$, a constant. With USE, because of (4.2), this becomes easier to satisfy.

Losses from transcoding Essentially, path decoders represent topological, as opposed to logical, knowledge of the CA. Their logical expression, when synthesized as sums-of-products of their input signals, cannot be simplified: they tend to contain the maximum number of possible coefficients. With p inputs and q outputs, the chip area of the decoder thus scales $O(p \times q)$. This means that USE reduces the width of the path decoder. Also according to our use cases, losses from transcoding scale well with increasing number of terminals.

Losses from control energy transport Let $W_{CP,TRANSPORT}$ and W_{DP} be the control-plane and data-plane bitwidth respectively. We want to assure that the loss from control energy transport $E_{xp_{ctl}}$ is not larger than the transport energy gained. From Eq. (3.3) and (4.3), the transport energy gained by EESC is

$$\begin{aligned} E_{xp_{gained}} &= E_{xp_{unsect}} - E_{xp_{sect}} = G_S E_{xp_{sect}} \\ &= G_S W_{DP} E'_{dyn} N_{cycles} \sum_{\forall s} \alpha_s l_s. \end{aligned} \tag{6.1}$$

We now argue that $E_{xp_{ctl}} < E_{xp_{gained}}$ as long as the number of resources (switches) is not too large. Assimilating the control plane to a wire section of width W_{DP}, activity factor α_{ctl} and length l_{ctl},

$$E_{xp_{ctl}} = W_{CP,TRANSPORT} E'_{dyn} N_{cycles} \alpha_{ctl} l_{ctl}. \tag{6.2}$$

The condition $E_{xp_{ctl}} < E_{xp_{gained}}$, can be written as

$$W_{CP,TRANSPORT} E'_{dyn} N_{cycles} \alpha_{ctl} l_{ctl} < G_S W_{DP} E'_{dyn} N_{cycles} \sum_{\forall s} \alpha_s l_s. \tag{6.3}$$

Because of (4.2), this is equivalent to

$$|R| < \frac{G_S W_{DP}}{(1 - \eta_{UE}) \log_2 |S_\mathfrak{r}|} \frac{\sum_{\forall s} \alpha_s l_s}{\alpha_{ctl} l_{ctl}}, \tag{6.4}$$

where $|R|$ is the number of resources.

If all activity factors are taken to be intrinsic and it is recognized that data and control sections cover the same distances on the surface of the chip, then, not knowing $\sum_{\forall s} \alpha_s l_s$ and $\alpha_{ctl} l_{ctl}$, we can nevertheless assume that they are of the same order of magnitude. Therefore the condition becomes

$$|R| < |R|_{max} \approx \frac{G_S W_{DP}}{(1-\eta_{UE}) \log_2 |S_{\mathfrak{r}}|}, \tag{6.5}$$

meaning that the number of resources must be smaller than some critical number, which increases with sectioning gain and with UEE. The exact value of $|R|_{max}$ is not known. It can be determined from a combined topological/statistical analysis of application and CA. A typical value we will encounter (in Section 8.1.3) is $|R|_{max} = (0.82 * 72)/(0.19 * 3) = 103$.

Decoding loss Decoding losses were not yet quantified. With many switching resources and a large useful-state set, they could become excessive. Conceivably, the better other costs in the control plane are optimized, the more decoding will become dominant, becoming the largest cost factor. Only a combined topological/statistical analysis can determine the conditions under which this happens.

6.3 Conclusions and Future Work

In this chapter, we have established a design pattern for the EESC control plane that follows six stages: compiling (including scheduling), splitting, fetching, transcoding, transporting and decoding. Costs of control can be attributed to each stage, and scaling properties with many terminals quantified, although the decoding stage is not yet analyzed at the far end of scaling. The design pattern can be used for fixed and variable-bandwidth communication architectures, and with implicit or explicit path specification.

Trade-off charts Mentally, we have often approached the optimization problem in diagrams with 4 dimensions, like the trade-off chart in Fig 6.5. We trade off design-time effort (path-finding, USA, and various tasks associated with compiling), including even the effort in developing the programs to accomplish this, against the 3 physical dimensions

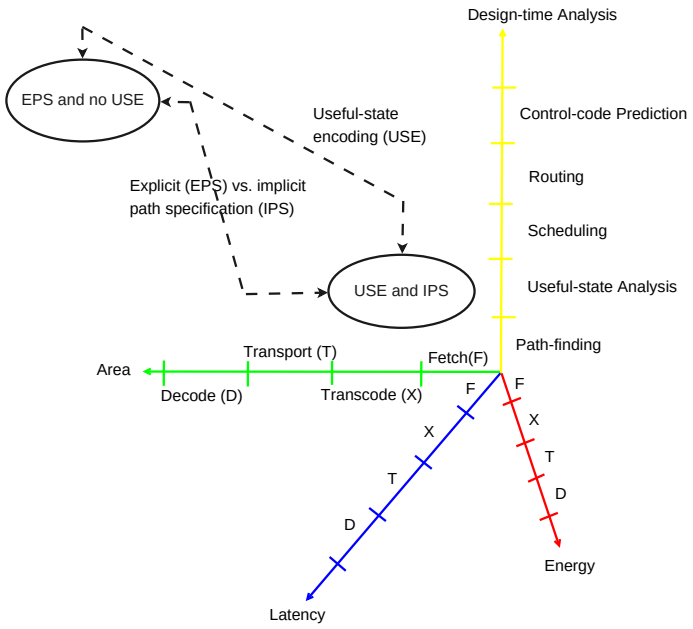


Figure 6.5: Trade-off chart of design-time effort against the 3D-space of area, latency and energy. Each dimension in the 3D-space has the same components, linked to physical control stages.

area, latency and energy. The physical dimensions have the same components: fetching, transcoding, transporting and decoding, which are correlated. (Splitting has no cost, only gain in design time.) Efforts in diminishing the losses associated with one or a combination of those stages brings other losses to the fore: if fetch and transport losses are reduced, as they were over the course of this work, a combination of transcoding and decoding losses becomes dominant. Since transcoders scale well, the prime cause of concern are the decoders in the switches. They are, with USE, larger than the default type of decoder used without it, and there are many of them. Implicit path specification, another example, saves physical costs from fetching, as well as costs in design and development time (for the compiler). For fixed-bandwidth CAs, this seems a pure advantage. Even for variable-bandwidth, the extra cost is only in scheduling, with no costs for (or problems with) prediction.

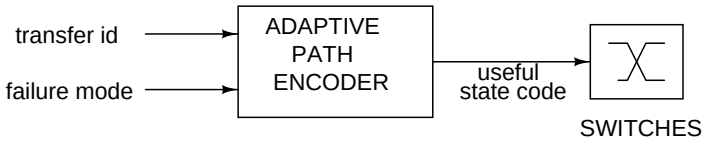


Figure 6.6: Adaptive path decoder.

Code optimization A dimension of optimization is left to address remaining problems of transcoding and decoding stages: the choice of the values of useful-state codes. Optimization of the control plane by selecting the actual values of control codes can have one of two purposes: Frequency analysis of useful states can lead to Huffman-type low-entropy encoding. This benefits control transport costs. The number of wires remains the same, but the control transport energy is better, since frequently occurring codes have a low bit count. Control code values can be chosen for easy decodability, benefiting energy, area and latency costs of decoding. To do so requires a statistical analysis of the transitions between useful states.

Survivability Interconnect reliability degradation in SoCs, mentioned in Section 1.1.2 may force designers to adopt built-in topological redundancy, such as advocated in Grover [41] for telecommunication networks. Grover finds the cost of redundant mesh-based networks intrinsically lower than that of ring-based networks. Control costs, however, are higher.

Adaptive Path Decoder Survivability of DSM SoCs could possibly be assured by integrating an adaptive path decoder into the control plane.

The problem of interconnect reliability degradation in SoCs, mentioned in Section 1.1.2, can be solved, as far as control is involved, with an adaptive path decoder, depicted in Fig. 6.6. Given an ISA which contains a communication instruction to announce a failure mode to the CA, presumably a single-link failure (although multiple link failures and switch failures can be accommodated), the lookup table of an adaptive path decoder can be constructed as follows. Calculate a full PSLT for the CA described by topology and the original, full, path-set. For each failure mode, eliminate the paths that contain the failed link from the path-sets and reduce the resulting PSLT to a TSLT. Construct the path decoder's lookup table from the combination of all TSLTs. The

most probable failure modes are yielded by an analysis described in [51]. Failure is detected by a controller described in [50]. Multiple failures raise the issue of complexity of analyzing all backup paths; this is not part of the present work, but illustrates the need for still further development of USA.

Chapter 7

Methods for Analysis, Simulation and Design

Though this be madness, yet there is method in't.

Hamlet, *Act II, scene 2.*

This chapter explores different approaches to analysis, simulation and design, their weak and strong points, and tools based on these approaches. Then we envisage a design flow for an EESC platform, and the data structures to be exchanged by designers, including the interface between a variable-bandwidth communication architecture and a topology-aware compiler.

We will also explore the merits of topology design, in the context of on-chip communication.

BEFORE before approaching design of a control plane, we should be able to simulate and analyze its operation.

7.1 Analysis and Simulation

The methodology for analysis and simulation changed over the duration of our work, as focus shifted from explicit to implicit path specification, from fixed-bandwidth architectures to variable-bandwidth, and as useful state analysis was developed. The first tool developed was a *control-code analyzer*. The approach was statistical: it used an abstraction of the application, furnished with physical (geometrical and elec-

trical) data, to study and predict the properties of the control-code flow. Communication architectures were fixed-bandwidth. The second tool was a *simulator*, applied to a proper communication-aware processor, embedded in a SoC. The communication architecture was still fixed-bandwidth, even single-transfer; simulation remained based on statistical and geometrical data. The third toolset used a *topological approach*. It employed the concept of useful-state encoding (USE), made abstraction of geometry, calculated figures of merit (UEE and ISG) and was suitable for variable-bandwidth CAs. The underlying assumption was schedule-neutrality, so compiler and application were left out of the model.

Statistical analysis of EESC In statistical analysis, we make use of the following entities: a compiler, application source code, a model for the processor, a geometric and circuit model of the data and the control plane, physical data on the semiconductor technology used, and a model for power consumption on a wire section. We must *(a)* determine the useful-state set, *(b)* perform power-aware placement, *(c)* determine the lengths of wire sections, *(d)* run a transaction-level (TL) simulation, *(e)* record the frequencies of useful states and the activity factors for the sections, and *(f)* calculate the savings and losses involved. Most of the statistical analyses described in Chapter 8 did not actually employ useful-state analysis, because we did not yet have it. It would have been easy to do so, anyway, since USA is trivial for fixed-bandwidth architectures.

Topological analysis In topological analyses, abstraction is made of specific applications, section lengths, useful-state frequencies, and activity factors. This is schedule-neutrality. It is recognized that only a part of the sectioning gain is intrinsic to the CA; this is the part that we study in topological analysis. The intention is to determine whether control for EESC is feasible for the CA and what ISG can be expected as a minimum, realizing that a statistical analysis after power-aware placement will yield a better sectioning gain.

Combined statistical/topological analysis The assumption of schedule-neutrality serves to separate the properties of application, including geometry of the tile, from the design of the control plane. Under this assumption one cannot, however, study all costs in our control plane

model: the energy costs of the transcoding and decoding stages are determined by transitions between useful states, not only by the frequency of their occurrence. Given the results of USA, it should be possible to record the frequencies of occurrence of useful states as well as their transition frequencies. This is the Markov transition matrix. Both frequencies of occurrence and of transition depend on the application, but perhaps even more on the scheduling performed by the compiler. We do not have a topology-aware compiler, but we can sketch the following modus operandi: *(i)* An ISA determines the properties of the CA. *(ii)* A path-set lookup table (PSLT) is determined for the CA under consideration. *(iii)* The path decoder and switch decoders are synthesized, using for instance a sum-of-products description. *(iv)* This synthesis determines the size of path and switch decoders, and their costs in latency. *(v)* Given the properties of compiler and application(s), frequencies are recorded. *(vi)* The transition frequencies of data and useful states determine the balance of transport energies between data and control plane. *(vii)* The transition frequencies between useful states determine the energy costs of path and switch decoders.

It is elaborate and time-consuming to have to actually compile and run the application to obtain useful-state frequencies, and it hinders fast design space exploration. The challenge of combined statistical/-topological analysis is to determine figures of merit on the basis of ISA and compiler scheduling properties alone.

7.2 Analysis and Simulation Tools

Each toolset developed over the course of experimentation consists of many lines of code, and uses different approaches in software engineering. To describe their concepts in detail would take up too much space; this section therefore contains only some salient features for each.

7.2.1 Segmented Bus Analysis

Segmented bus analysis, a statistical method, was employed in the context of intra-tile communication for a hierarchical memory system, designed by means of data storage and bandwidth exploration (DTSE) [25]. In experiments of this type, a memory architecture optimization tool (Atomium/MA (Memory Architect)) [9] is used to design the hierarchy. The functional units in the tile are identified from inspec-

tion of the C code. The communication architecture is fixed-bandwidth, consisting of a number of "buses" (shared media). Scheduling is performed by another tool, Atomium/SBO (Storage Bandwidth Optimization). "Bus sharing" methodology, described in [113], is used to define a shared-media architecture suitable to satisfy the bandwidth requirements. The connection of individual memories to buses is then synthesized on the basis of matching of the bus connection structure to the memory architecture by means of a transfer conflict graph (TCG).

After power-aware layout, described in [46], all bus segment and control code line-lengths are extracted using a commercial routing tool. The source of control is deemed to be a location at the center of the floorplan.

A program named `sba`, i.e. segmented bus analysis, which is in fact, a control-code analyzer automates analysis of instrumented source code and the profiled run of an application. The method can yield valuable information on the distribution of control codes, but it cannot yield an exact bit stream to be used for explicit path specification, since it is based on a high-level (C-code) mapping of the application made by Atomium, not on the properties of the SoC, or the scheduling of the proper compiler. Moreover, a degree of uncertainty is present in the control codes due to register indirect addressing. Finally, since it is based on profiling, the method must estimate the actual data energies being spent on the buses. (Profiling records events, not data.) This limits the accuracy of the energy balances obtained from this approach.

7.2.2 System C Power Simulator

To address these deficiencies, a simulator was developed to obtain the actual bitstreams in a simple SoC. System C enables us to zoom in on areas of interest of different level and granularity, ranging from RTL level to communication instruction execution and topological properties. The simulator uses the CP paradigm, and operates in the context of a memory hierarchy developed using Atomium and power-aware floor-planning, as described above. The CA is single-transfer; the topology is a linear segmented bus. Analysis is still statistical. The control plane uses implicit, not explicit path specification.

Modeling the processor Since we want RTL simulation of the CA, but are unwilling to sacrifice many cycles for the detailed simulation of a

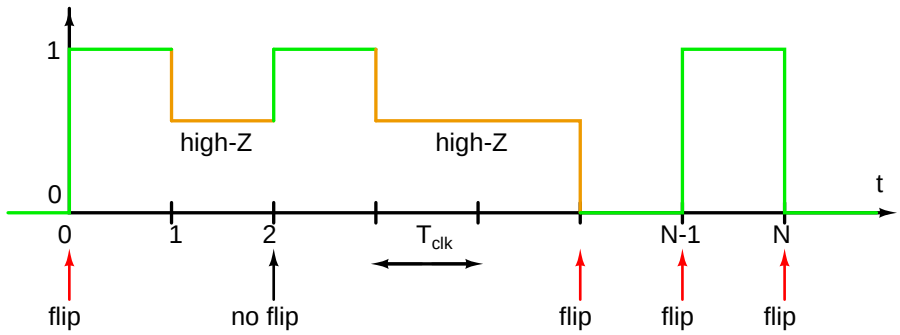


Figure 7.1: Power simulation for 4-state-signals. Signals that transit from 0 via Z to 0 or 1 via Z to 1 have not changed and do not contribute to any energy budget.

processor, we use a TL model for the processor. The didactic DLX processor modeled in System C by Grunwald [43] was found to be suitable. In our simulator, the model for the processor is general and high-level. It uses the 5 classic processing stages of computing (fetch, decode, execute, memory access, write-back) and passes on the full state of the machine between those stages. The interface between processor and CA is independent on any particular processor hardware, making the CA model itself portable to other processors. The DLX model comes with a compiler, a `gcc` 2.7 derivative, and an assembler written in `perl`, easily modifiable for our needs. This is necessary to incorporate extra communication instructions (like for loop buffer control) in the ISA.

Energy- and topology-aware classes in System C System C knows the 4-state signals of CMOS. The four states are 0, 1, Z (high-impedance) and X (invalid).

Using our model, two concerns arise: *(i)* System C allows for RTL simulation, but it is not a power simulator, and *(ii)* even with a simple topology like a simple linear sectioned bus, debugging the simulator program proves to be difficult.

Wire segments can at times not be driven at all or, at other times, active CMOS drivers can have their outputs shorted. Fault conditions in the network, caused by incorrect control codes driving wire section buffers, cannot be traced by classic debugging techniques. They propagate over the network, and are observed at places in the simulation where they did logically not occur.

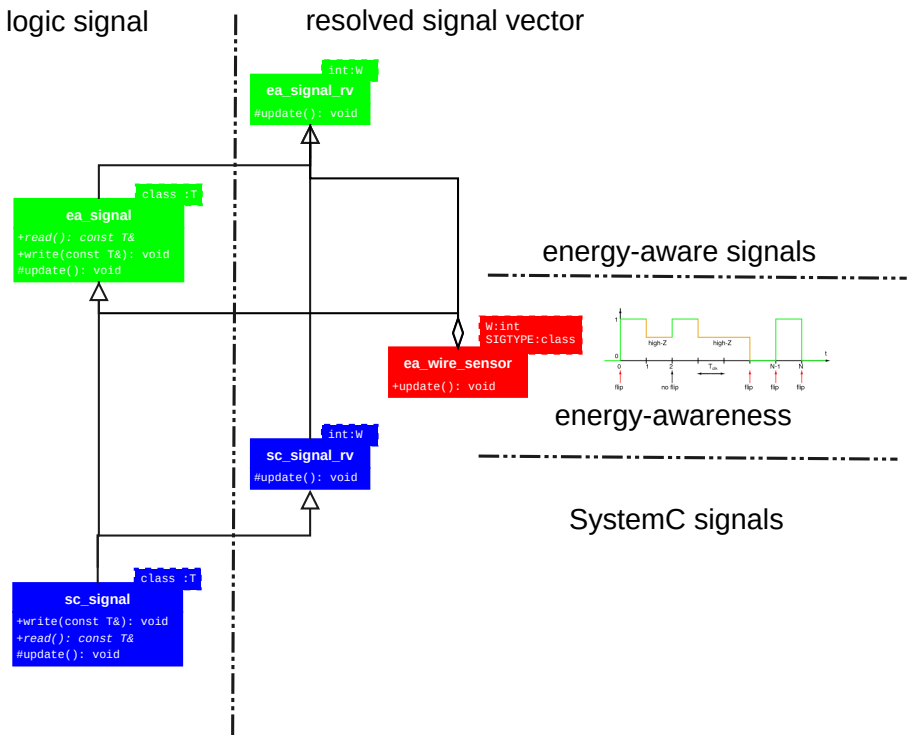


Figure 7.2: Energy-aware classes. The native classes for signals and resolved vectors of signals are superclassed by energy-aware versions containing a wire sensor which observes signal states and accumulates energy dissipated on the wire.

Inside the System C power simulator, modified classes are used, termed energy-aware resp. topology-aware, for signals and components. Unified Modeling Language (UML)-diagrams (depicted in Figs. 7.2 and 7.3) show how the new classes relate to the ones being overloaded.

Energy-aware signals have the behavior depicted in Fig. 7.1, and can be used to measure power on sectioned wires. Signal flips, costing energy, are transitions from 0 via Z to 1 or from 1 via Z to 0. Signals that transit from 0 via Z to 0 or 1 via Z to 1 have not changed and do not contribute to any energy budget. To address problem *(ii)*, we can provide all components in the SoC under test, which are terminals in the CA, with the capability of asserting the continuity of the paths in the topology and the validity of the signals. The latter assertions are formulated in terms of Property Syntax Language (PSL) [96]. PSL is not part of Sys-

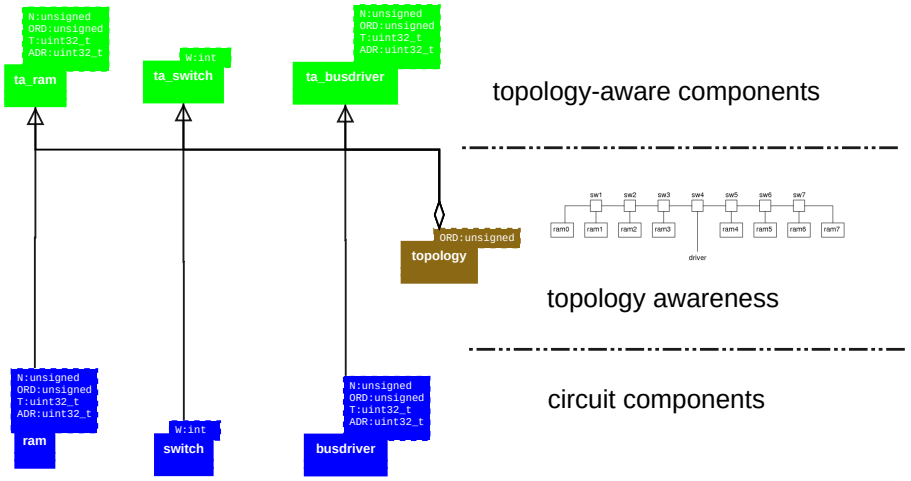


Figure 7.3: Topology-aware classes. Key components of the simulator, RAM, switches and load/store unit (`busdriver`), are superclassed by topology-aware versions, which assert the correctness of signal state and path continuity, enabling a report of fault conditions when and where they occur, before propagating.

tem C as it is of other HDLs, but a suitable C++ PSL-library was written by the author. The topological knowledge of the network, a linear shared bus with drop-off sections has also been encoded. Components that have built-in knowledge of topological relations, and even proper time sequences to be asserted during simulation, are termed *topology-aware*.

7.2.3 Useful-state Analysis Tools

A third toolset is primarily intended for design using USA, and features a topological approach. It fits in the design flow recorded below, and is suited for variable-bandwidth CAs. A brief description of the tools' usage forms is given in Appendix A. The tools are aimed at finding the all-paths set in a meshed topology, constructing a path allocation graph out of a CA description, and a path-set lookup table from a path allocation graph.

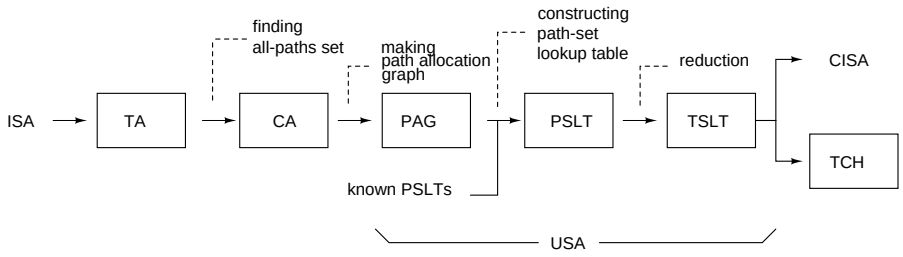


Figure 7.4: Data structure dependencies in USA.

7.3 Design Flow

7.3.1 Dependencies

The data structures involved in control plane design have some dependencies, shown in Fig.7.4. It is informative to observe how these structures are exchanged between the actors in the design process.

1. An instruction set designer defines the instruction set architecture (ISA). The ISA is an extensive specification that including transfer and operational instructions for the processor, their addressing modes, the communicating entities, and, under our assumptions, the level of concurrency. For the network designer this implies the terminal arrangement (TA). The terminal arrangement should be seen as a set of constraints on communication, but not yet as the full description of a communication architecture (CA).

2. The network designer develops, after exploration, a network to match the TA. Below, we will briefly describe the type of considerations that come into play. The design of the network, and the terminal arrangement, result in a formal CA: a combination of a topology and a set of useful paths.

3. The network designer finds the useful-state sets, attributes control codes to each, and, if the ISA is not topology-aware, reduces the PSLT to a transfer-set lookup table (TSLT). This is useful-state analysis (USA). Byproducts of useful-state analysis are optimum encoding and the data for eventual switch decoder synthesis.

4. The CISA (communication ISA, including communication-aware instructions, if needed) is now fixed. The transfer-compatibility

hypergraph, derived from the TSLT, serves as a specification to the memory architect and the writer of the scheduler. The compiler, in case of explicit path sspecification, needs to know the actual control codes, in order to issue them.

The transfer-compatibility hypergraph satisfies the minimum requirements of the ISA and TA. Opportunities to schedule more transfers than the TA required, may exist. These would follow from topology design. In order to utilize those, the ISA would have to be adapted. Alternatively, the network designer can reduce the CA more, reducing the size of the control plane in the process.

7.3.2 Topology Design

There is a large body of theoretical research on optimal topologies [29] in the data plane, based on graph-theoretical metrics such as vertex and edge symmetry, degree δ, average distance d_a, network diameter D, and bisection width B_C, among others. These parameters have a direct impact on network performance. Fig. 7.5 shows some basic candidate topologies. The choice of such a topology, is often central in the decisions that determine performance of supercomputers, as witnessed by the choice of a fat cube topology for distributed shared-memory NUMA machines like the SGI Origin 3000 [29] or of the K-ring for the Swiss-T1 cluster [36]. We want to argue that the criteria for optimality are different for SoCs than for high-performance computing (HPC), and not really explored yet.

The prime topological constraints in HPC come from the subdivision in modules, chips, boards and chassis required in a physical system. The parameters of topology and packaging technology determine the placement of communication nodes in the system, as well as the channels' bandwidth. (For an instructive exploration of this, we refer to [29], Chapter 3). In SoCs, these constraints are relaxed, because all connectivity is on-chip. The cost elements of an on-chip CA are area consumed and congestion caused in the upper metal layers of the chip, while merit is found in the ability to provide many parallel high-bandwidth links and topological redundancy (to combat interconnect reliability degradation). These requirements both point in the direction of meshed topologies. In fact, a non-cyclic topology like a linear bus can never guarantee topological redundancy.

Whereas in HPC the vertices in topologies like those of Fig 7.5 are

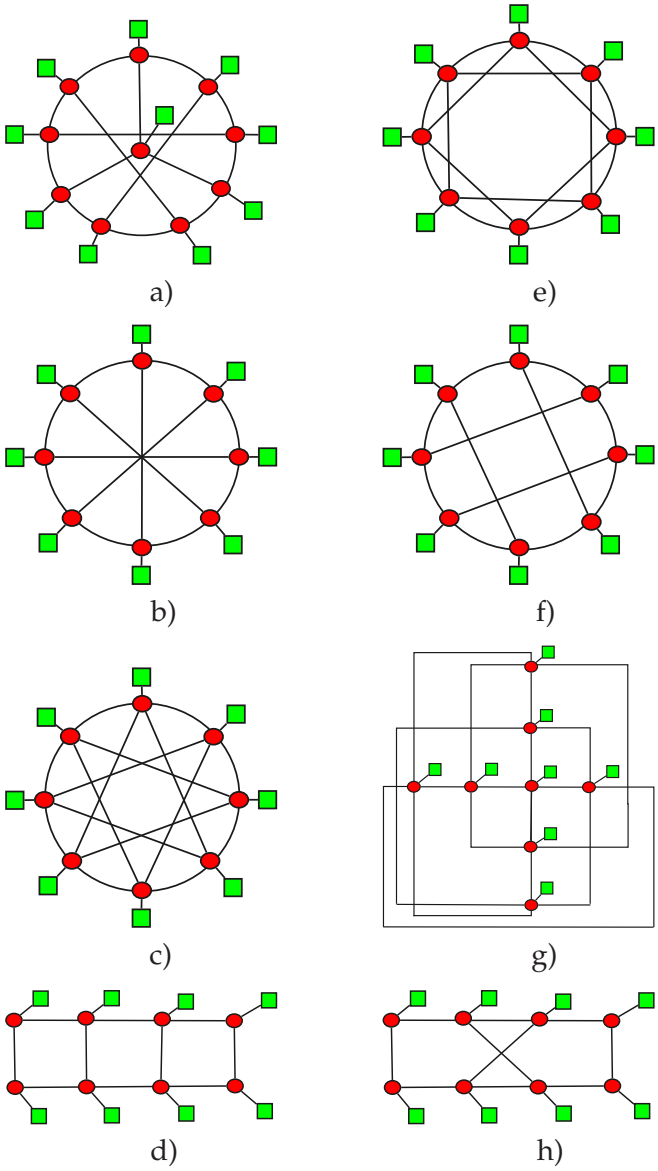


Figure 7.5: Eight generic networks from high-performance computing, with 8 (or 10) terminals: a) Moore graph, b) Octagon, c) K-ring d) Ladder e) Midimew, f) Hypercube, g) AMP h) Twisted Ladder. (Source: [87])

routers and the links are serial, in SoCs the vertices are atomic switches and the links parallel wires. Other fundamental differences between

the intra-tile domain in SoCs and HPC include: *(i)* the opportunities of 3D stacking in SoCs, reducing congestion; *(ii)* the importance of the terminal arrangement. In HPC the terminals are often functionally identical, while with SoCs they belong in most cases to distinct terminal classes with different roles; and *(iii)* the prospect of centralized compile time scheduling within SoCs, in contrast to the semi-random routing process of HPC.

While we have, in this section, raised more questions than we answered, we think that USA can contribute to the study of aspects *(ii)-(iii)* above. Indeed, our topological use cases in Chapter 8 yield at least one more concept of merit for CAs that USA can calculate: the *histogram of concurrency of transfers* over the useful states, given a certain terminal arrangement. This amount of concurrency determines the opportunities for the scheduler and also provides a measure for the redundancy built into the CA.

7.4 Future Work

In future work, it must be possible to re-introduce scheduling by means of *combined statistical/topological analysis*, while still foregoing geometry and possibly, the application. Such an approach would focus on control states rather than on data activities, and allow us to determine the feasibility of control over EESC at the far end of scalability, with very many switching resources, where transcoding and decoding losses dominate over transport and fetching losses. Precise knowledge of data values is not essential in the control plane, but puts a heavy burden on the simulation.

Chapter 8

Use Cases

The purpose of computing is insight, not numbers.

Richard Hamming.

This chapter presents experiments conducted and published while our framework was developed. Two designs for control planes were analyzed statistically, the second one twice (once without and once with USE). A Digital Audio Broadcasting (DAB) receiver design was analyzed in order to study the feasibility of EESC control. A design for a Global System for Mobile Telecommunication (GSM) speech encoder, was optimized once with loop buffers and once using USE.

We will derive from these experiments some rules of thumb on fixed- and variable bandwidth communication architectures, and conclusions on our figures of merit. After the statistical analyses, we will present topological analyses on various CAs ranging from simple to complex: an intra-processor functional-unit interconnect, a CA with a broadcasting terminal arrangement, and various architectures with interesting topology, including grid and torus interconnects often found in SoCs. We conclude the chapter with a conclusion on the scalability of control of EESC with many terminals.

HAVING driven the design framework now as far as it will presently go, we turn to the use cases that guided its development. Benchmarks are taken from the field of low-power processing for multi-

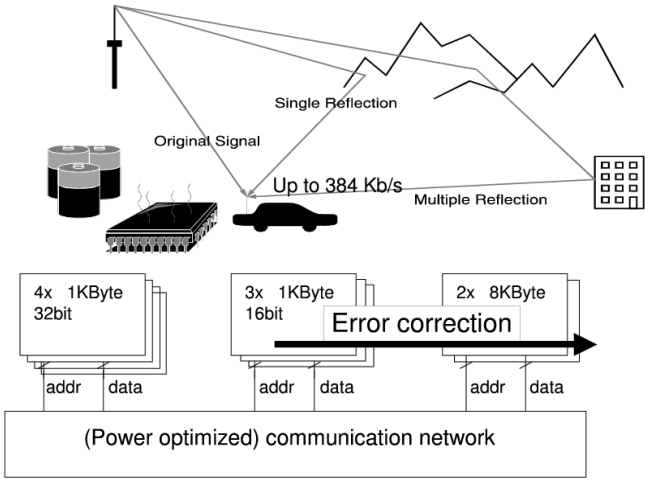


Figure 8.1: Principles of DAB receiver application. At the receiver, the Orthogonal Frequency Division Multiplex (OFDM) is reconstructed by a forward 256-point FFT. A frequency and time de-interleaver unscrambles the input symbols and a Viterbi decoder performs error correction based on redundant information.

media.[1]

Fixed-bandwidth systems are simple, and easy to control or schedule. Their energy-saving properties are found to scale well with bus network multiplicity and number of terminals. They can easily be used for memory hierarchies with register-indirect memory-addressing. Variable-bandwidth CAs are harder to schedule than fixed-bandwidth CAs. They are found to still have good sectioning gain. They are usable as long as the control path W_{CP} is narrow enough at all section points. Whether this can be achieved, is decided by useful-state analysis.

8.1 Statistical Analyses

8.1.1 DAB Receiver

In our publication [57], we performed control-bit analysis by processing a profiled run of a DAB receiver application, of which the principles are pictured in Fig. 8.1, As implemented, the receiver has three functional

[1] We should not exaggerate the importance of the multi-media aspect: from the viewpoint of control, it is difficult to characterize an application by the data it carries.

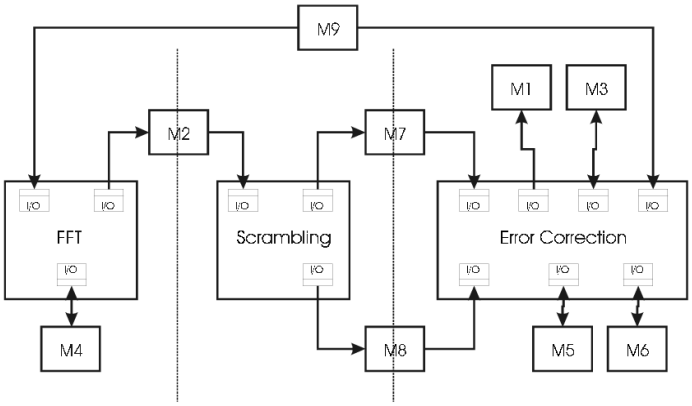


Figure 8.2: The DAB receiver architecture.

units; the interconnection is shown in Fig. 8.2: a FFT subsystem (`f1` in the diagrams below), a Viterbi decoder or error correction processor (`f2`), and a de-interleaver or unscrambling processor (`f3`).

To estimate the required switch control power consumed on a sectioned bus for different solutions in the floorplan of a data-intensive application, we made 4 different physical floorplans for a DAB receiver. Each represented a different trade-off between power efficiency and circuit area, and had a different on-chip memory count and consequently, a different complexity of the sectioned bus structure. Two memory partitions (sets of memories of different types): `m1` and `m3` are referred to in Figures 8.3-8.6, which shows the four solutions. In each solution, the optimization of putting more arrays in different and smaller memory modules (named `m1-1`, etc.) can be pursued to a different extent.

After data storage and bandwidth exploration (DTSE) [25] analysis of the problem, 4 different optimizations are chosen, to set 4 alternative tasks for the design process. The solutions are 4 designs that all feature 3 parallel buses, but have a different number of memories: respectively 4, 8, 10 and 12. They all feature an 8-bit bus (`b1`), which in some solutions is extended to include some 16-bit sections, and two 32-bit buses (`b2` and `b3`), as shown in Figures 8.3-8.6.

Activity-aware floorplanning [46] minimizes the length of sections with high activity. Since the sections most frequently used are minimal in length, this technique globally optimizes the energy consumed by the data transfer over the sectioned buses. Fig. 8.7 shows the four

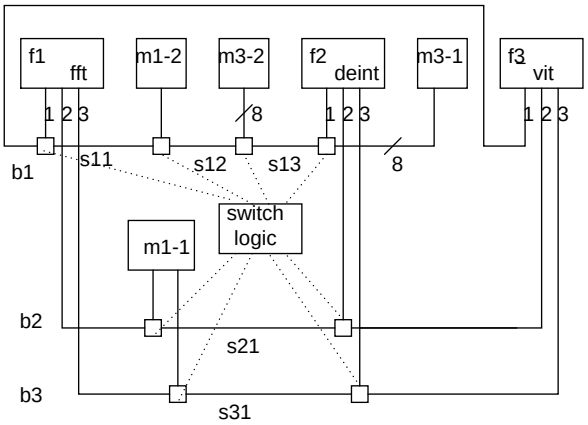


Figure 8.3: Four possible solutions were obtained from design space exploration, with 4, 8, 10 and 12 memories respectively; each has 3 functional units (`f1` = FFT, `f2` = error corrector, `f3` = descrambler), and 3 buses. This is solution 1, with 4 memories.

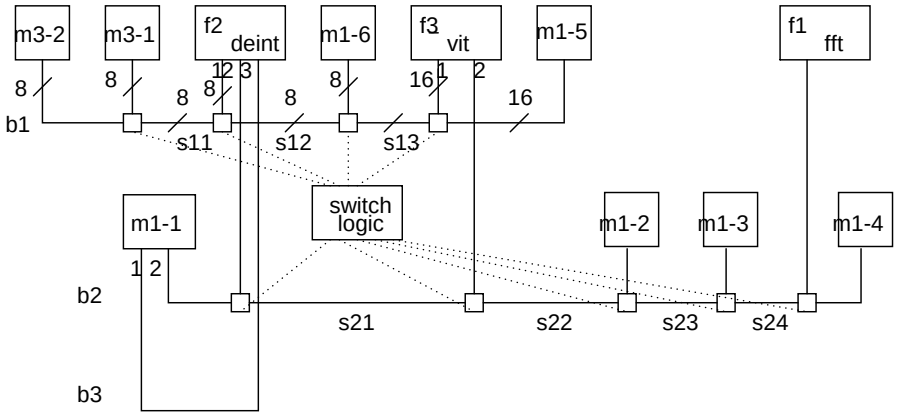


Figure 8.4: Solution 2 from design space exploration, with 8 memories.

layouts that result from each optimization.

The experiment allows following observations to be made:

- A typical control bit activity pattern is similar to the one depicted in Table 8.1, from solution 3, the 12-memory design, where FFT-processor `f1` is active and uses buses `b1` and `b2`:
- Looking at Fig. 8.6, we see that only switches `b1:sw8`, `b1:sw9`, `b2:sw3` and `b2:sw4` are active, switching the right-hand side of

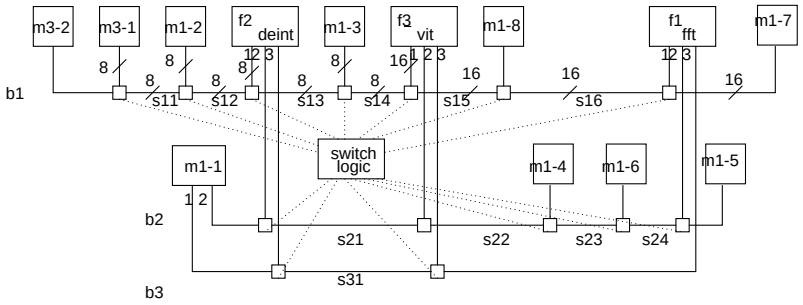


Figure 8.5: Solution 3 from design space exploration, with 10 memories.

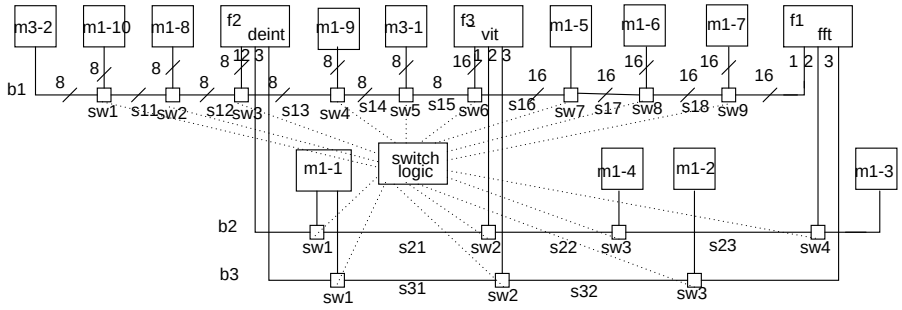


Figure 8.6: Solution 4 from design space exploration, with 12 memories.

Table 8.1: Typical activity pattern of control bits. The control codes are for 6W3T switches with default decoder type.

```
                    b1
 sw1  sw2  sw3  sw4  sw5  sw6  sw7  sw8  sw9
1100 1100 1100 1100 1100 1100 1100 1100 1000
1100 1100 1100 1100 1100 1100 1100 1100 1100
1100 1100 1100 1100 1100 1100 1100 1000 1010
1100 1100 1100 1100 1100 1100 1100 1100 1000
1100 1100 1100 1100 1100 1100 1100 1100 1100
1100 1100 1100 1100 1100 1100 1100 1000 1010

       b2                        b3
 sw1  sw2  sw3  sw4          sw1  sw2  sw3
1100 1100 1100 1100         1100 1100 1100
1100 1100 1001 0111         1100 1100 1100
1100 1100 1100 1100         1100 1100 1100
1100 1100 1100 1100         1100 1100 1100
1100 1100 1001 0111         1100 1100 1100
1100 1100 1100 1100         1100 1100 1100
```

buses `b1` and `b2`. The period is 3 cycles, in this case.

Localization of switch activity would encourage us to seek for clusters of groups of switches, that can be efficiently driven from

Figure 8.7: Four possible solutions for the layout: 4, 8, 10 and 12 memories.

loop buffers. We find that if two switches change data direction, all switches in-between also change data direction and show activity. This is detrimental to the locality of switching activity. The effect, observed also at other times and on other buses, made us think of clustering by bus rather than by locality. The inevitability of thhe global correlation of control code patterns is confirmed in another use case, in Section 8.2.1.

Using the switch control patterns and the line-lengths, we can calculate energies. In Fig. 8.8, the energy consumed on the buses by the DAB application is compared with the energies that would be consumed were the bus not sectioned. In the first place, the comparison corroborates the advantage of a sectioned bus from the point of power efficiency.

Comparing the data energy on the bus not-sectioned with the energy consumed on a sectioned bus, we see that both $E_{sectioned}$ and $E_{unsectioned}$ reach a minimum which is not radically different between the four solutions. This indicates that sectioning does not impose different targets for physical layout optimization than a non-sectioned solution.

In Fig. 8.9 we compare the energy required to transport switch con-

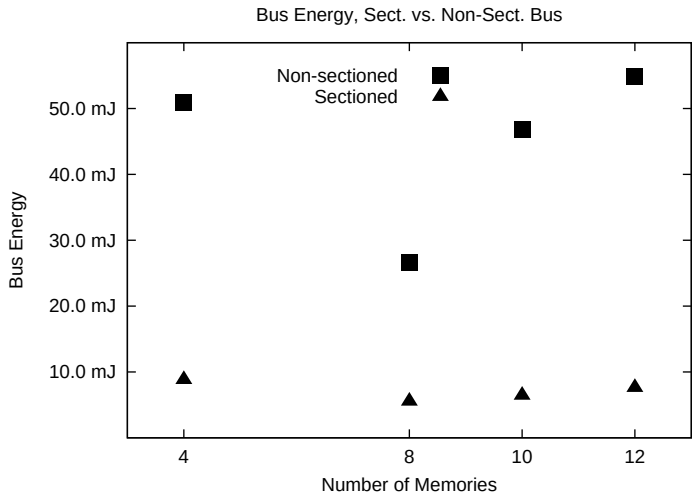


Figure 8.8: Comparison of sectioned vs. non-sectioned energy over the application, for four DAB receiver design choices

trol bits with the data energy over the sectioned buses, for all design choices. We find that it is of a lower order of magnitude. Intuitively, we attribute this to a good choice of the switch codes, chosen in this experiment to minimize the activity on the control bit wires. More importantly, our experimental set-up contains many more active data and address lines than control bit lines. Also there is only limited activity on some buses in some branches of the program, thanks to activity-aware placement. This observation is the core of the reasoning made on p. 115 in Section 6.2.2: the best way to limit costs of transport of control, when compared to costs of data transport, is keeping the number of control wires down. Furthermore, we see that:

- Power consumed on the sectioned bus for transport of data is 17-21% of the power consumed on the equivalent unsectioned bus.
- Power consumed by transport of control bits is much smaller than the saving obtained from sectioning in the first place. It is in the range of 1.5-6% of the data transport power, after sectioning.[2] So

[2]Some inaccuracy here comes from the denominator of the comparison: the power for data transport is dependent on the content of the data and addresses, which can only be guessed at if only profiling but no platform compilation or full hardware simulation is done.

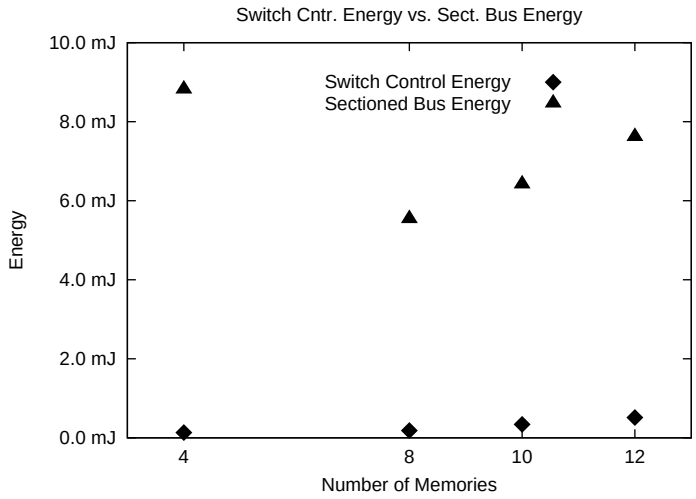


Figure 8.9: Comparison of switch control energy vs. sectioned bus energy over the application, for four DAB receiver design choices

the reduction of the control power is quite satisfactory.

- Clustering the switches may make sense both locally and per bus; it is hard to decide from this experiment. Often the switches do not change state because the bus is not in use. At other times, frequent patterns can be seen on a section of a bus because only short sections are being used.

- Switching occurs often, and for long periods, every cycle. This follows from what Atomium/SBO's optimization considers to be a cycle: a period through which accesses to external memory are scheduled. Consecutive cycles during when internal registers are accessed, are not counted. Only if the access schedule would be completely the same for two successive cycles, w.r.t. sources and sinks as well as data direction, would there be no switching activity, or else when the bus were simply not in use. The communication architecture is thus reconfigured frequently.

Conclusions from this experiment Not knowing at the time the full extent of design space, the experiment yielded no strong optimum for sectioned bus energy. Basically, looking at control transport energy

only (we did not consider other classes of loss in this experiment), the experiment showed that the case for the sectioned bus had held well. Observation of the control bit flow indicated that we should not go for optimization of the transport energy component, but instead try to minimize the energy cost of fetching the control information. The concept of a pure control bit analyzer proved dissatisfying, since it could not always predict the exact flow of control bits in a real SoC, where one error would disable the application. From this, the CP paradigm evolved, and the concept of implicit path specification.

8.1.2 GSM Speech Encoder with Loop buffers

Our paper [59] demonstrates control of a low-power communication network by a simple processor as shown in Fig. 8.10. It had only one linear energy-efficient sectioned bus (LESB). The paper explores the energy cost saving of switch control transport by inserting clustered loop buffers in the control processing path, thereby exploiting periodicity of the control bit flow. It describes the implementation and simulation of such a system, in a simple but significant test case, concentrating on programmed control and the loop buffers. Also, from these results, it considered the prospect of expanding the EESC control concept to multiple buses including non-linear control topologies.

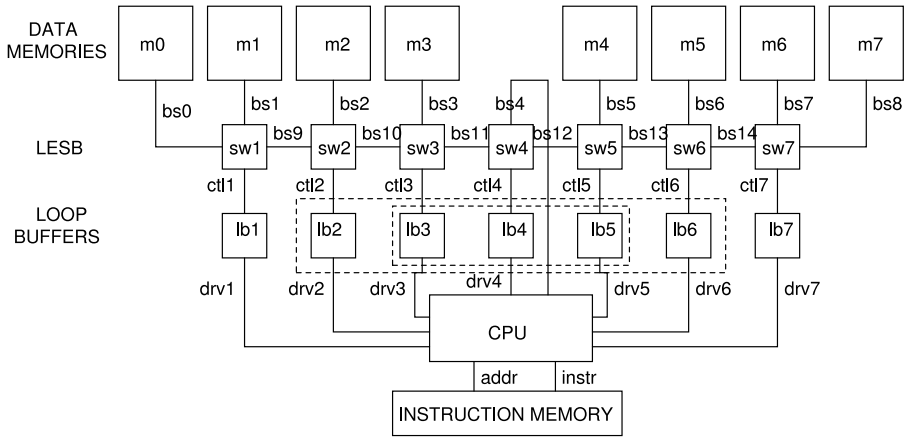


Figure 8.10: Simple communication processor with 1 LESB and 8 memories.

The full platform is shown in Fig. 8.10. It includes a DLX [55] CPU, with instruction memory, a single linear sectioned bus, 8 distributed data memories, sectioning switches, control wires and up to 7 loop

buffers. This is enough to implement a sizable application. It is in fact a template, one we eventually wanted to expand to 16 and more memories.

In the CP paradigm we need to define the "communication instructions", the extra instructions that our processor needs. The DLX instruction set in fact already contains a "communication instruction" which expresses the memory and register addressing modes that the processor needs. It is not necessary to invent new communication instructions for this purpose. The realization of this fact leads ultimately to the concepts of implicit path specification and the path decoder. The Load/Store instruction is adopted as a communication instruction. The instruction gets an extra meaning: that of additionally implying the value of the switch control bits. In addition, we have a new Loop Buffer instruction in one of the two forms `swlbon` n s and `swlboff` n. n is a selector for individual loop buffers or clusters of them. s is the length of a loop that the program performs.

Our benchmark performs GSM 06.10 lossy speech compression [44]. It encodes frames of 16-bit PCM voice samples at a rate of $8kHz$ into GSM frames. Profiling it, we find that 70% of the cycles are spent calculating the codec's long term predictor (LTP) parameters. This routine has a loop sequence, containing in fact a sub-loop that was already unrolled, of 80 communication instructions, repeated 80 times. For the purpose of testing loop buffering, exploiting the temporal redundancy of this single loop can attain nearly all improvement that is possible by temporal re-use. The buffers need to be 128 cycles long (128 is the smallest power of 2 exceeding 80).

After code analysis, it is straightforward to order the memory modules optimally along the bus, and to maximize the activity on the short wires. As a result, the short and most active wire sections are centrally located on the bus, between $sw4$ on one side and $sw3$ or $sw5$ on the other side. From the topology now follows that the switches $sw3$, $sw4$ and $sw5$ are the most active and that their activity is heavily correlated. It is thus clear that clustering is most likely to help for that group of switches. The group of switches $sw2$-$sw6$ is another candidate for clustering.

For the purpose of energy computation, we assume a technology node of 130 nm. Energy-aware layout minimizes the length of the most active wires and yields the length of the control wires involved. Memory sizes and energy consumption by wires and memories are based on

CACTI 4.2 [22] and Banerjee [13].

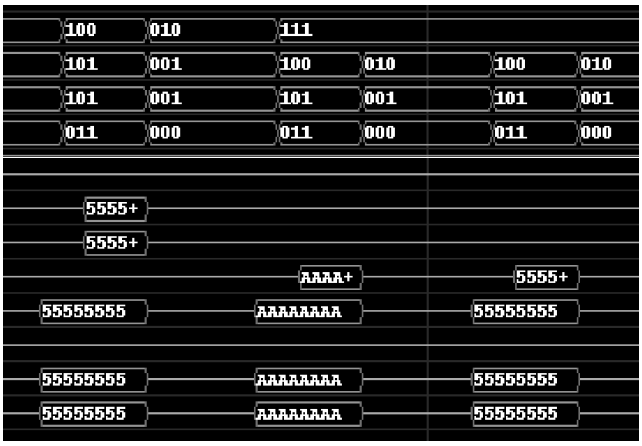


Figure 8.11: GSM speech encoder: sectioned operation. The control codes (above), driving the sectioning switches, ensure that data wires (below) are only driven when carrying information. At other times, the drivers are open-impedance.

In the simulator, already described in Section 7.2.2, we want to simulate all aspects at functional level for the processor and at RTL level for the bus. Fig. 8.11 shows a trace of the control signals as they drive the sectioning switches, and the alternation of the bus sections between active and high-impedance state. The simulator accumulates the voltage flips on the wires and the energy costs of reads and writes of all memories. It verifies correct operation verification statements on topology-aware System C objects. Energy measurements by the simulator are confirmed by running an exhaustive set of elementary tests, simple enough to be verified by inspection.

Observations on design choices We are now able to compare the efficiency of several alternatives: not sectioning the bus, sectioning without loop buffers, introducing a loop buffer per switch, and clustering loop buffers. A cluster of triple loop buffers must necessarily be better than any other loop buffer clustering solution, since almost all cycles are spent accessing the small memories centrally located on the bus. Because the loop buffer's SRAMs are narrow, which is relatively power-inefficient, clustering makes good sense. Thus we do not, in the end, deploy separate loop buffers per switch, but only one loop buffer for a

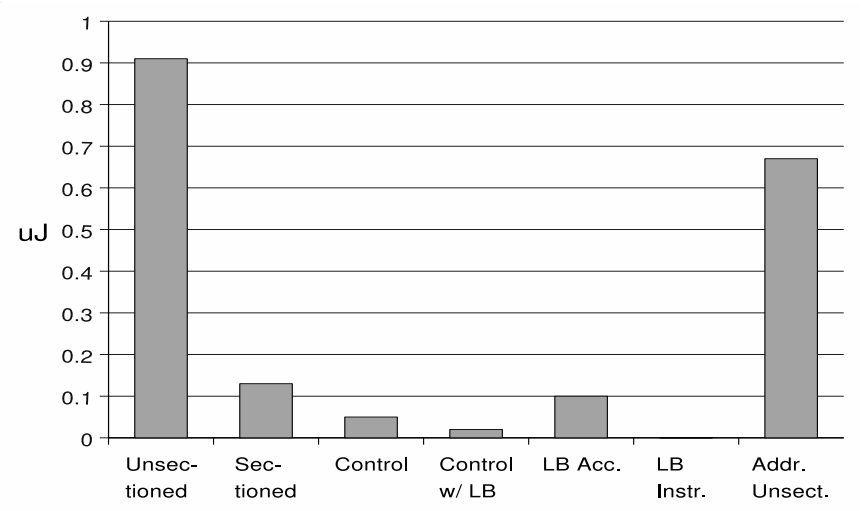


Figure 8.12: GSM speech encoder results: transport and access energies.

cluster of 3 switches, shown in Fig. 8.10 within a dashed box. Fig. 8.12 shows the results of our experiments. Without sectioning the transport energy of data on the bus is $0.910\mu J$, for the length of the code sequence involved. After sectioning, it is $0.127\mu J$. We conclude that

- Improving energy-efficiency by sectioning works: in sectioned operation, even without loop buffers, the data transport energy is reduced by 86% at a cost of 4.7% in switch control transport energy. The sectioning gain[3] is 81%. The switch control transport energy does not need to be minimized further, at least not for a linear bus.
- On the other hand, the bus address transport energy, shown for reference, is 73% of the data transport energy before reduction, and in this experiment not reduced by the sectioning. There may exist other techniques to reduce address energy for low-power SoC [78, 80], but that should not stop us from optimizing it at the level of wire sectioning as well. Hence, the address bus should also be sectioned in the future.
- Loop buffer operation reduces switch control transport energy to $18.2nJ$ from $46.7nJ$, but incurs high costs in read accesses for the

[3] In [59], the definition of sectioning gain differed slightly from the definition in this work. Rather than re-calculating the data, we quote figures from the paper.

loop buffers ($99.8nJ$). Sectioning sectioning gain reduces to only 73%. Loop buffer access losses outweigh the saving obtained. Loop buffers are no panacea to further reduce control transport energy, if that were still required.

We learn here that we should not confuse between transport loop buffering and fetch loop buffering. The latter is effective in reducing the fetch energy of *all* instructions, since it is built into the fabric of the processor. Our transport loop buffers are 'out on the surface' of the chip, have no connection to the program counter, and can save nothing but transport energy. But their memory access overhead is too large for that.

- Data memory accesses and the transport energies towards the instruction memory are both costly. This is normal, since we did not optimize the platform under test in any way for this.

- Fetching Load/Store instructions requires $563nJ$. This figure is only meaningful in order of magnitude. Since the instructions must be fetched anyway (implicit path specification), they are not included in the budget. Also, our simulator is in no way designed to optimize the cost of instruction fetches. But the fact does indicate that explicit specification would have been prohibitively costly.

- Fetching the Loop Buffer On/Off instructions, on the other hand, requires only $0.136nJ$. It happens only seldom, and on its own this is very efficient.

Conclusions from this experiment The paradigm of the CP, first used in this experiment, proved helpful in identifying all energy costs. The simulator lets us verify the operation of EESC and measure all energy costs, including some not measured before, and make a complete energy balance. The balance of control vs. transport energy remained favorable. Further reduction of control energy proved again, as in the DAB experiment, not to be needed. We reduced the overhead in communication instruction fetch energy to near zero, by the suitable means of implicit path specification. Controlling EESC, we did not squander the savings made by sectioning. We identified the essential component of the communication processor: the path decoder. Loop buffers proved not effective in this context. As for scalability, we decided that

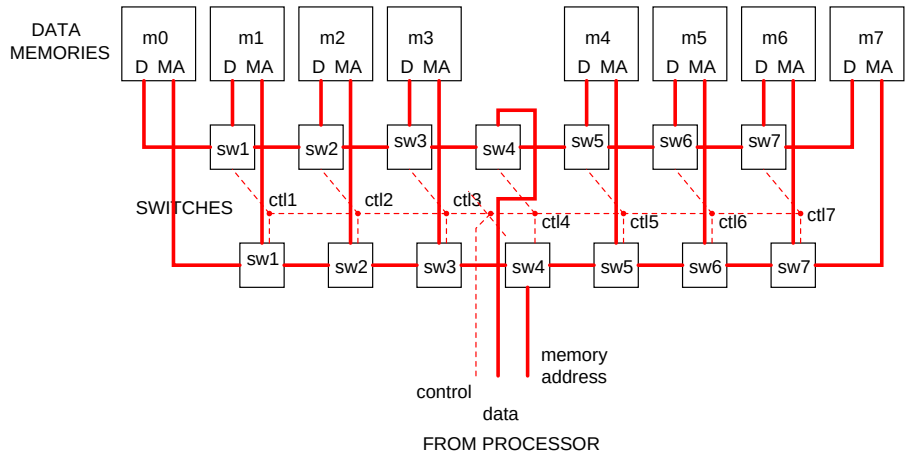


Figure 8.13: A simplified template for a single-transfer-per-cycle CA with 8 scratchpad memories and sectioned linear bus network topology.

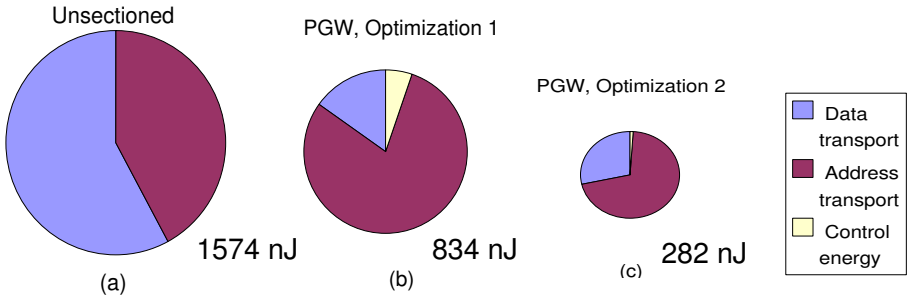


Figure 8.14: Statistical analysis for GSM speech encoder: (a) transport energies without wire sectioning, optimized without useful-state analysis, (b) transport and control energies with EESC, optimized without USE, and (c) same, but optimized with USE, memory-address bus network sectioning and sub-word selection.

the path decoder mechanism would still serve us with up to 16 modules and up to four linear buses, which was at that time our next immediate concern, and that we had resolved the issue at the bottom-end to medium range of complexity, which was significant as it covers all single-issue processors with single linear buses.

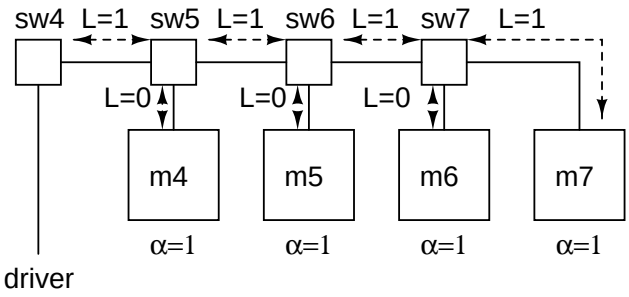


Figure 8.15: ISG calculated for a half sectioned linear bus.

8.1.3 GSM Speech Encoder with Useful-state Encoding

For our paper [62], we revisited the CP with one linear energy-efficient sectioned bus (LESB), using the same GSM speech encoder [44] program and architecture now simplified to Fig. 8.13. This second optimization employed USE, memory-address bus network sectioning, and sub-word selection. This reduces the width of the control plane: UEE is 81%, which accounts for a large drop in control energy. Fig. 8.14 shows the data plane (data and address) transport energies for the unoptimized platform (a), as well as the results of EESC after the first (b) and second (c) optimization. The total pie areas in Fig. 8.14(a-c) are proportional to the data plane transport energy: 1574 nJ, 834 nJ, and 282 nJ. The pie-charts show the ratio to control energy of data and address transport energy respectively. The latter consists almost entirely of control transport energy. Sectioning gains after each optimization are 47% and 82 % respectively. After the second optimization, losses from energy transport of control bits have almost completely disappeared: they drop from 42 nJ to less than 3 nJ. Interconnect losses with EESC are in the end dominated by losses from transport on the address bus network. The schedule-neutral ISG for this CA is 37.5%, implying that the program-specific sectioning gain goes from 10% in the first optimization to 44% in the second. This understates the effect of the first optimization. Related to data transport energy alone, excluding address transport energy, program-specific sectioning gain is 38% for the first optimization. We also find that switch selection by subword and address bus sectioning are indispensable.

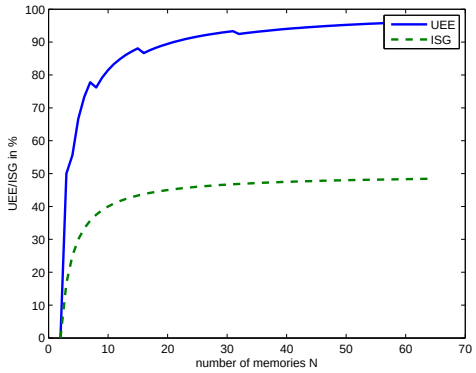


Figure 8.16: UEE and ISG for CA with linear topology.

8.1.4 Fixed-Bandwidth Architectures

We expand our fixed-bandwidth example of Fig. 8.13, to the general case of a fixed-bandwidth CA with M bus networks, of linear topology, and N memories. The load/store unit f_0 takes part in every transfer. The data plane has data and memory address sub-buses. With shared media, path specification means bus specification. The path decoder is multiplexed per bus. For each bus, control of data and memory addresses is the same. Data and memory switches are co-located. Checking the figures of merit, we find, by averaging the lengths of the unused sections over all useful states (cfr. 8.15) that schedule-neutral ISG $G'_{IS} = (N/2-1)/N$. Finding the useful states is trivial with CAs with a linear topology and a single transfer per cycle: there exist $1+2\times N$ useful states. Useful-state control code transport requires $\lceil \log_2(1+2\times N)\rceil$ bits, not $3 \times (N-1)$. (3 is the default number of wires for a default switch, $N-1$ is the number of switches). UEE is $\eta_{UE} = 1 - \lceil \log_2(1+2\times N)\rceil/(3\times(N-1))$. ISG and UEE are plotted in Fig. 8.16. For 8 memories, $\eta_{UE} = 76\%$; for 16 memories, $\eta_{UE} = 87\%$. Useful encoding uses only 24,% respectively 13% of the number of control wires that would be used without USE. The limit for very long buses is 50% for ISG and 100% for UEE. The merit of useful-state encoding is not limited by the size of the communication architecture; gain from wire-sectioning has a bound, which comes essentially from the symmetric nature of the topology.

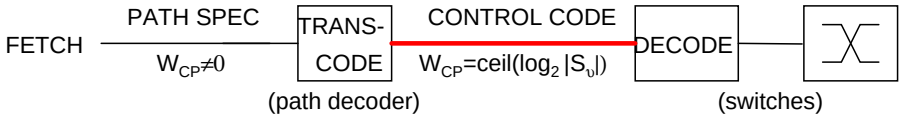


Figure 8.17: Control plane design pattern for implicit path specification.

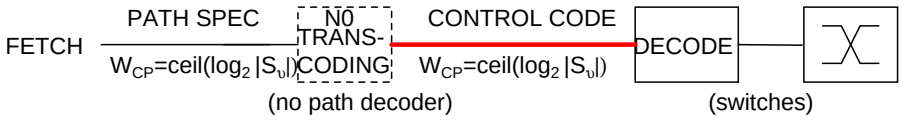


Figure 8.18: Control plane design pattern for explicit path specification.

8.1.5 Rules of Thumb on Path Specification and CA Bandwidth

Control planes with implicit transfer set specification follow a general pattern shown in Fig. 8.17. They do not incur fetch stage penalties, as all information for the path transcoder is already present in the instruction, even when no EESC is employed. The exception to this is path selection information, which is why we write $W_{CP} \approx 0$ and not $W_{CP} = 0$ at the point of fetching. With USE, the path decoder is of minimal width. With explicit path-set specification, the control plane does not need a path decoder, but must fetch useful-state codes explicitly for each transfer instruction, as shown in Fig. 8.18.

The path for control transport is of minimal width, because control encoding has minimal topological redundancy. Useful path codes are presented to all switches. This reduces wire congestion, which is beneficial. With USE, the switches' decoders can be larger than they would be without USE. Optimization by choosing the numerical values of the control code after combined topological/statistical analysis could reduce the decoding losses.

Tables 8.2 and 8.3 summarize rules of thumb that can be used for design space exploration of ISA and data-plane topology, to determine the consequences for the control plane.

Table 8.2: Synopsis of use cases, for implicit path specification

Stage	M single-transfer CAs with N terminals	variable-BW CA
Fetch	none	none
Transcode	$area : O((\log_2 N)^2)$ $delay : O(M)^a$	$area : O((\log_2 \lvert S_\mathrm{u} \rvert)^2)$
Control Transport	$\lvert R \rvert < \lvert R \rvert_{max} \propto \frac{G'_{IS}}{1-\eta UE}$	
Decode	Code optimization (statistical)	

[a] Or bus-pipelined.

Table 8.3: Synopsis of use cases, for explicit path specification

Stage	M single-transfer CAs with N terminals	variable-BW CA
Fetch	$O(\log_2 M + \log_2 N)$	$W_{CP} < K$
Transcode	none	none
Control Transport	$\lvert R \rvert < \lvert R \rvert_{max} \propto \frac{G'_{IS}}{1-\eta UE}$	
Decode	Code optimization (statistical)	

8.2 Topological Analyses

Variable-bandwidth communication architectures are intended for special needs, message passing, broadcasting, direct addressing, non-uniform memory architectures, topological optimizations, process variability issues or other requirements for path redundancy. They are harder to schedule than fixed-bandwidth communication architectures. They normally still have good sectioning gain. They are usable as long as the width at all sections in the control path is low enough. Whether this can be achieved is decided by useful state analysis.

Without loss of generality, we can focus on single-component CAs, since disconnected bus networks can be analyzed separately.

8.2.1 Functional-Unit Interconnects

Functional unit interconnects are likely to contain directional paths, have many switches and a high correlation between switch positions. Because of the correlation, they should have good UEE. Because of the many switches, they need it.

The terminal arrangement for a functional-unit interconnect is obviously very dependent on the exact ISA. For example, we take a typical intra-tile network: the functional unit chaining interconnect described by F. Barat Quesada in his thesis [15]. This network has already been seen in Fig. 3.11, but is now treated in more detail. Many VLIW processors contain register bypass networks. CRISP has an extra facility for functional unit chaining: two functional operations can be chained within a single execution cycle. For this purpose, the delay of the data path operation has been kept to less than half the clock cycle. Also, the functional units are augmented with a shunt register, which can be bypassed when results are needed within the current clock cycle. Results needed within the current cycle, but not any more afterwards, are termed *ephemeral results*. Barat Quesada has not implemented it using EESC. Our own implementation with EESC is not meant to be optimal: it is a seemingly logical first try. The network topology is shown pictorially in Fig. 8.19 and mathematically in Fig. 8.20. It is influenced by "linear bus" thinking, since we know as yet very little about optimal intra-tile topologies.

Communication-architecture analysis From the terminal arrangement (TA) and the network topology (there are no meshes), we can get the all-paths set and the useful-paths set. To deduce the terminal arrangement, we need to know the ISA. In CRISP, functional instructions have a destination and two sources. To encode the instructions that produce ephemeral results, a constant register is used as destination. Such registers have a fixed value which is, for instance, always zero. Writing to such register has no effect. If the source for an instruction is an ephemeral result, instead of a register, this is specified by `#n`, where `n` is the number of the functional unit that produced the effect. An example is given below.[4] Instructions issued in the same cycle are joined by the C-language logical-OR notation `||`; subsequent cycles are separated by

[4] In this example code, `a1`, `a2`, etc. are address registers and `d1`, `d2`, etc. are data registers, not to be confused with the terminals `d1`, `d2`, ... in Fig. 8.19.

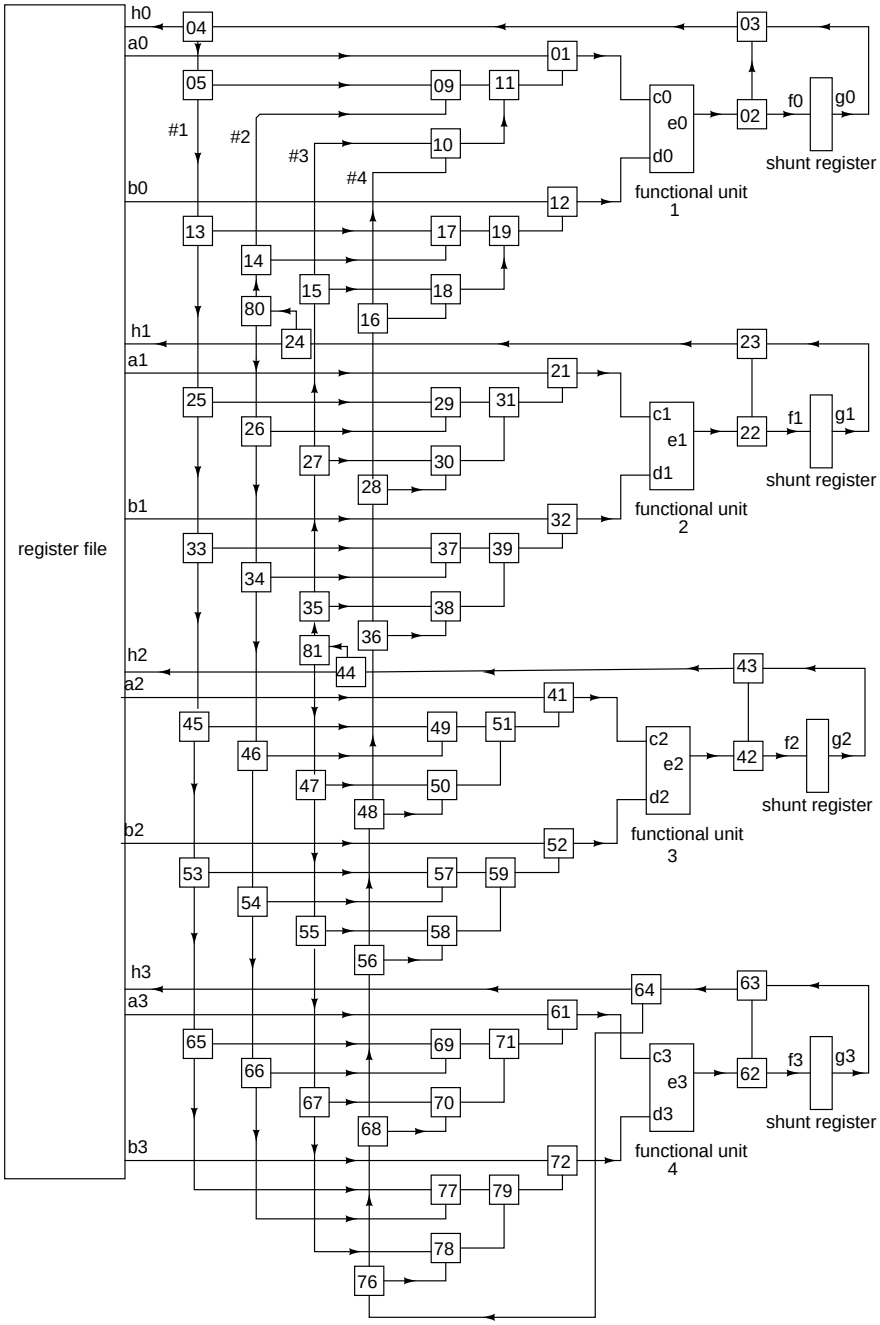


Figure 8.19: Functional unit chaining: functional unit network from CRISP.

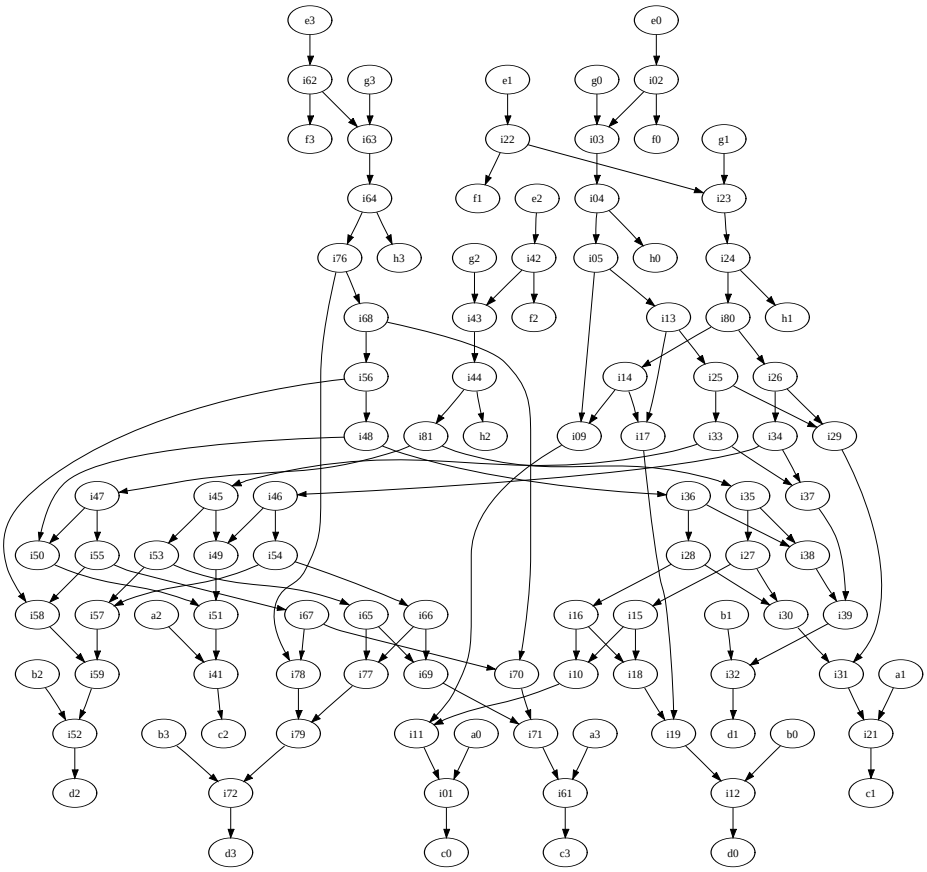


Figure 8.20: Functional unit chaining: network topology.

semicolons.

```
   SHL.1 d2, d0, d1 || MUL.3 d8, d6, d7 ;
   ADD.1 NULL, d3, d4 || ADD.2 d9, #1, d5 ;
   ADD.1 NULL, d9, a1 || ADD.2 NULL, d5, a2
|| SUB.3 NULL, #1, #2 || ADD.4 d13, #3, a3 ;
```

In this example, functional unit chaining takes place once in cycle 2 and thrice in cycle 3. The useful paths used in cycle 3 are shown in Fig. 8.21. A functional unit cannot be chained with itself; the ISA does not allow it. The code snippet below, using plain register bypassing, is however possible, yielding still more useful paths:

```
   SHL.1 d2, d0, d1 ;
```

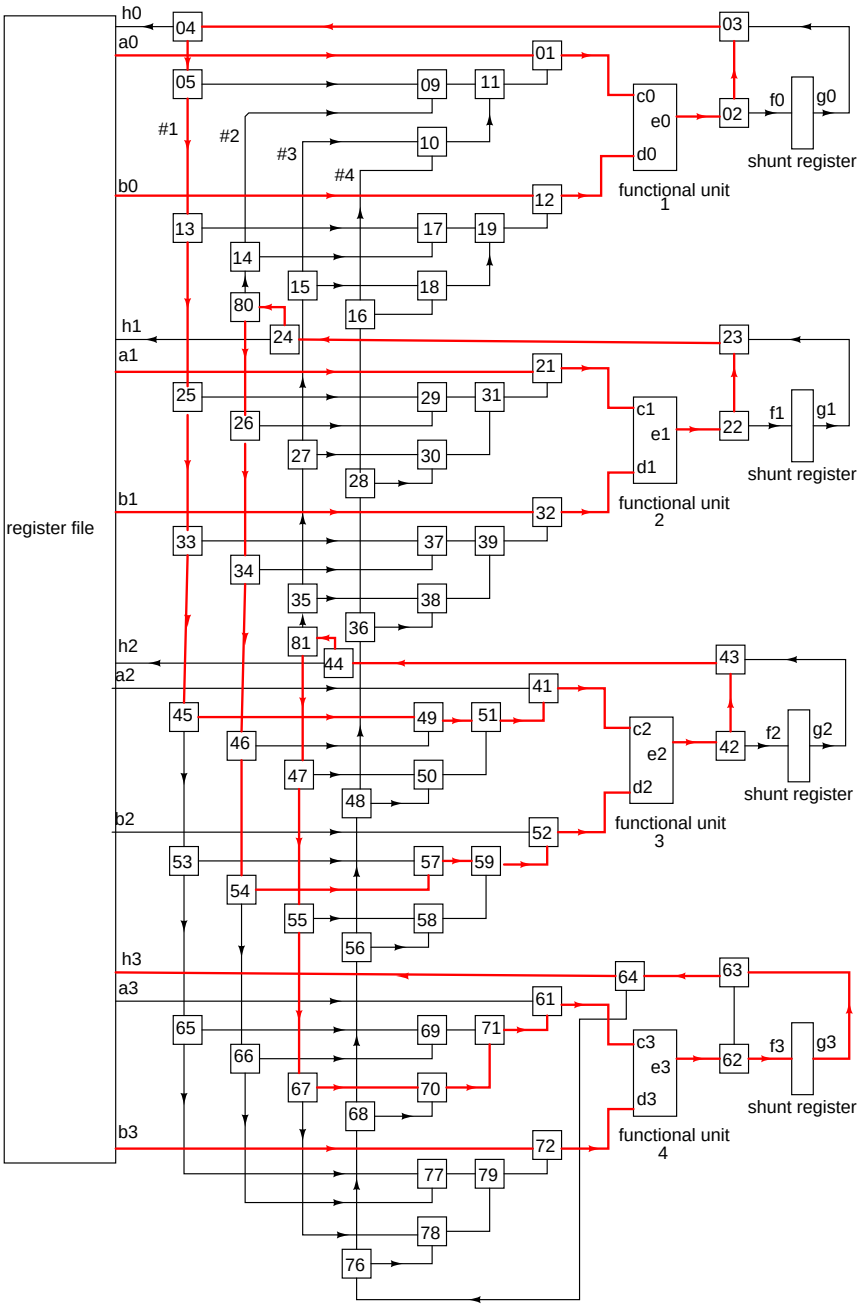


Figure 8.21: Functional unit chaining: useful paths used in cycle 3, shown in red.

```
ADD.1 d9, f1, d5 ;
```

The useful-paths set (UPS), which follows from the TA, must provide round-trip paths from the output of a functional unit to its own inputs, like for instance, in Fig 8.19, the path *g0→01→03→04→05→09→11→01→c0* does. Altogether there are 72 useful paths, shown in Tables 8.4 and 8.5. This makes USA computationally challenging. We recognize 8 terminal classes in the TA and can identify them in Fig. 8.19. We have register file sources (2 classes) and destinations (1 class): 3 classes together; sources (2 classes) and outputs (1 class) of functional units: 3 classes; shunt register in- and outputs: 1 class each, 2 classes altogether. The CA has only a single component: it is completely connected. The network designer (the author) has in some way wanted to realize logical "buses": 8 for the inputs of the functional units, and 4 feedback networks. He might have wanted to create 12 fixed-bandwidth CA, but has in fact created one single-component CA, because the "buses" are interconnected, with a bandwidth that is probably variable. Surely, it can carry from 0 up to 16 concurrent transfers. This is intuitive, and the DIS algorithm confirms it: the concurrency number is 16 (not 12). A CA is fixed-bandwidth if all maximal independent path-sets are same-size. Besides, using USA, one can quickly find some maximal independent path-sets of size 12. Thus the CA is not fixed, but variable-bandwidth. The concurrency number is 16.

Alternate view With this particular CA, one can take a alternate topological view of resource allocation. Crucial resources can be identified, like for instance `04` (or `24`, `44`, and `64`, respectively). These control access to certain logical "buses", like, in this case, the four feedback "buses" `#1`, `#2`, `#3`, and `#4`. One can even group crucial resources in 12 critical resource groups, called `rg03`, `rg23`, `rg43` and `rg63` (each grouping 2 resources) and `rg02`, `rg12`, `rg22`, `rg32`, `rg42`, `rg52`, `rg62`, `rg72` (each grouping 4 resources). Resource group `rg03`, for instance, contains resources `03` and `04`. Resource group `rg02` contains resources `02`, `09`, `10` and `11`. The critical resource groups are shown in Fig. 8.22. Interestingly, when the states of the groups are combined (with a Cartesian product), demonstrably less than their all-state space is useful. Resource group `rg03` has only 3 useful states out of 4 possible; Resource group `rg02` has only 4 useful states out of 16. One could envisage an ad hoc design tool that enumerates the useful-state

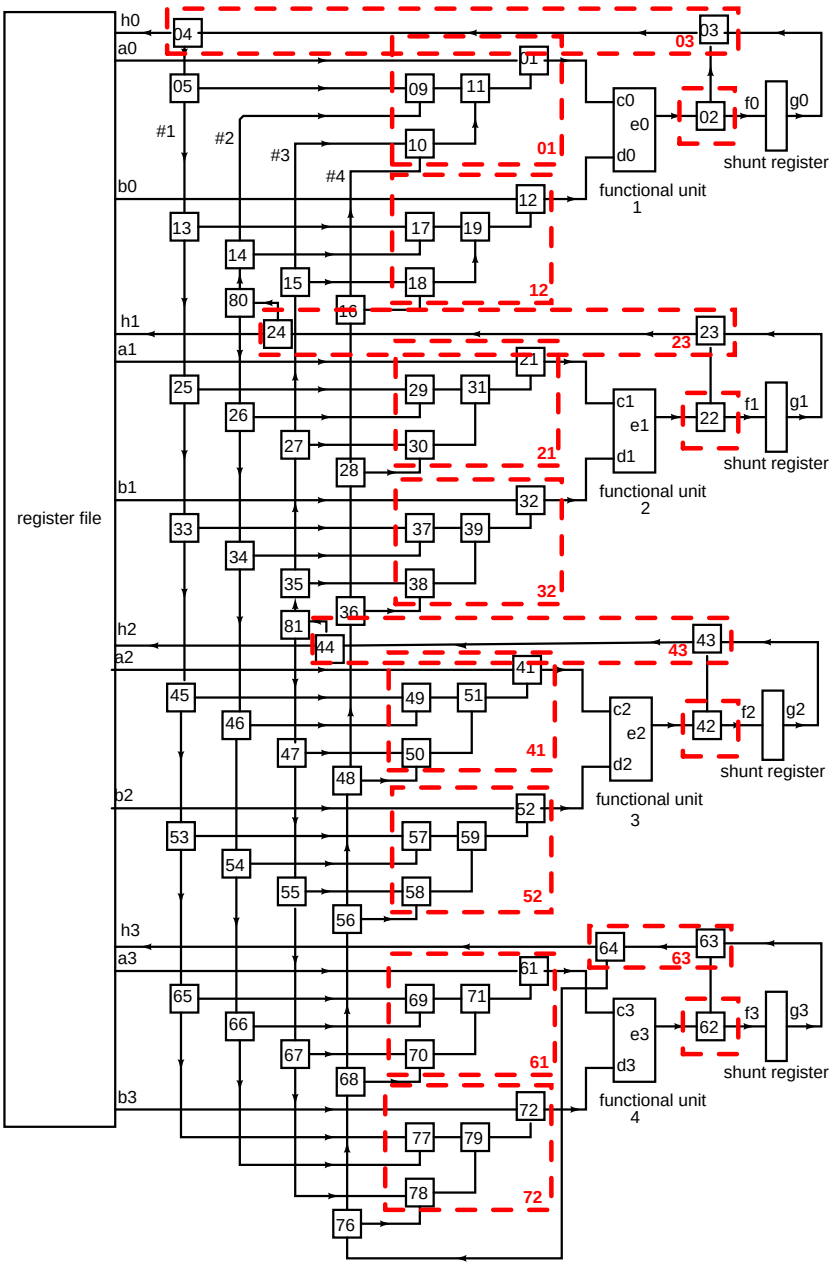


Figure 8.22: Functional unit chaining: 12 critical resource groups identified, and shown in red. Note that resources 02, 22, 42 and 62 are also crucial, though not grouped.

Table 8.4: Useful-paths set for functional unit chaining, part 1.

```
a0->01->c0     a1->21->c1     a2->41->c2     a3->61->c3
b0->12->d0     b1->32->d1     b2->52->d2     b3->72->d3
e0->02->f0     e1->22->f1     e2->42->f2     e3->62->f3

g0->03->04->h0 g1->23->24->h1 g2->43->44->h2 g3->63->64->h3

e0->02->03->04->05->13->25->29->31->21->c1
e0->02->03->04->05->13->25->33->37->39->32->d1
e0->02->03->04->05->13->25->33->45->49->51->41->c2
e0->02->03->04->05->13->25->33->45->53->57->59->52->d2
e0->02->03->04->05->13->25->33->45->53->65->69->71->61->c3
e0->02->03->04->05->13->25->33->45->53->65->77->79->72->d3

e1->22->23->24->80->14->09->11->01->c0
e1->22->23->24->80->14->17->19->12->d0
e1->22->23->24->80->26->34->46->49->51->41->c2
e1->22->23->24->80->26->34->46->54->57->59->52->d2
e1->22->23->24->80->26->34->46->54->66->69->71->61->c3
e1->22->23->24->80->26->34->46->54->66->77->79->72->d3

e2->42->43->44->81->35->27->15->10->11->01->c0
e2->42->43->44->81->35->27->15->18->19->12->d0
e2->42->43->44->81->35->27->30->31->21->c1
e2->42->43->44->81->35->38->39->32->d1
e2->42->43->44->81->47->55->67->70->71->61->c3
e2->42->43->44->81->47->55->67->78->79->72->d3

e3->62->63->64->76->68->56->48->36->28->16->10->11->01->c0
e3->62->63->64->76->68->56->48->36->28->16->18->19->12->d0
e3->62->63->64->76->68->56->48->36->28->30->31->21->c1
e3->62->63->64->76->68->56->48->36->38->39->32->d1
e3->62->63->64->76->68->56->48->50->51->41->c2
e3->62->63->64->76->68->56->58->59->52->d2
```

space based on the interdependency of these 12 resources groups, together with the equally crucial single resources `02`, `22`, `42` and `62`. To enumerate these would take $4^8 \times 3^4 \times 2^4 = 84934656 \approx 85.10^6$ steps. The task is not insuperable and all useful states can be derived from this. But the design tool would require much optimization in order to run in reasonable time, and can only be used for one CA. Instead of doing this, we will continue with regular USA.

Useful-state analysis The experiment allows the following observations to be made. We can construct the PAG and find the PSLT. The concurrency number is large (because we must include the paths from functional unit to shunt register, 16). The performance of MIPS, IPS and PPS were mentioned already in Table 5.2. MIPS does the job in 4 hours; IPS and PPS, took longer than 8 hours. In total there are 1385 directed maximal independent path-sets. The complete directed PSLT (or TSLT,

Table 8.5: Useful-paths set for functional unit chaining, part 2.

```
g0->03->04->05->09->11->01->c0
g0->03->04->05->13->17->19->12->d0
g0->03->04->05->13->25->29->31->21->c1
g0->03->04->05->13->25->33->37->39->32->d1
g0->03->04->05->13->25->33->45->49->51->41->c2
g0->03->04->05->13->25->33->45->53->57->59->52->d2
g0->03->04->05->13->25->33->45->53->65->69->71->61->c3
g0->03->04->05->13->25->33->45->53->65->77->79->72->d3

g1->23->24->80->14->09->11->01->c0
g1->23->24->80->14->17->19->12->d0
g1->23->24->80->26->29->31->21->c1
g1->23->24->80->26->34->37->39->32->d1
g1->23->24->80->26->34->46->49->51->41->c2
g1->23->24->80->26->34->46->54->57->59->52->d2
g1->23->24->80->26->34->46->54->66->69->71->61->c3
g1->23->24->80->26->34->46->54->66->77->79->72->d3

g2->43->44->81->35->27->15->10->11->01->c0
g2->43->44->81->35->27->15->18->19->12->d0
g2->43->44->81->35->27->30->31->21->c1
g2->43->44->81->35->38->39->32->d1
g2->43->44->81->47->50->51->41->c2
g2->43->44->81->47->55->58->59->52->d2
g2->43->44->81->47->55->67->70->71->61->c3
g2->43->44->81->47->55->67->78->79->72->d3

g3->63->64->76->68->56->48->36->28->16->10->11->01->c0
g3->63->64->76->68->56->48->36->28->16->18->19->12->d0
g3->63->64->76->68->56->48->36->28->30->31->21->c1
g3->63->64->76->68->56->48->36->38->39->32->d1
g3->63->64->76->68->56->48->50->51->41->c2
g3->63->64->76->68->56->58->59->52->d2
g3->63->64->76->68->70->71->61->c3
g3->63->64->76->78->79->72->d3
```

since only one path-set exists for each transfer set) has 543744 useful states. $UEE = 1 - \frac{\lceil \log_2 543744 \rceil}{72 \times 3} = 90.74\%$. With USE, we need 20 wires instead of 216 (since there are 72 switches). Intrinsic Sectioning Gain is unknown since our most optimized tool lacks the option to calculate it. It is evident that ISG will be high; Fig 8.21 indicates this, showing, for one particular useful state, that many wire sections are not driven.

It is instructive to compare the above solution to another design solution without USE, where path specification would have been explicit and transport loop buffers would have been employed. With loop buffer clustering per bus, we would presumably have needed 8 loop buffers, 4 of which would have to be clustered with the functional units, and 4 others with the functional unit feedback bus bars, since control codes for the switches along those are mutually heavily correlated but

not with individual functional-unit loop buffer's instruction streams. Loop buffer clustering per location would be impractical, since nearby switch states are evidently not correlated.

It might be possible to find a better topology for the network, but given the results of USE, this is not immediately called for. This example also illustrated how sensitive a communication architecture can be to the exact form of the ISA, and how little sense it makes to experiment with advanced topologies without reference to a terminal arrangement.

Tentative approach to address interconnect reliability degradation How to adapt this CA to be survivable for single link or node failures, is not evident. Protecting against single link failures, on long wires only, is feasible by making 16 long wires bidirectional and circular. In order not to overload the figure, we show in Fig. 8.23 only 7 of the 16 bidirectional connections. 32 additional 6W3T switches need to be introduced. Performing USA on the 16 individual single-fault scenarios, given the improvement of optimized design support tools, and the availability of parallel processors, is possible. We must note that backing up against failure by introducing rings is sub-optimal, according to [41]. Rings are easy to comprehend; mesh-based survivable networks in telecommunication have been found to be less costly. Our design tools are suited for both.

8.2.2 Other Variable-Bandwidth Architectures with Interesting Topology

Four variable-bandwidth architectures with all-to-all terminal arrangement Fig. 8.24 shows four common network topologies: a linear, circular, K-ring [74] and butterfly topology. Each has 8 terminals and the same terminal arrangement: all terminals can communicate with all others. The networks have 6, 8, 8 and 16 resources (switches) respectively. The linear, circular and butterfly networks use 6W3T switches, controlled by 3 bits each, while the K-ring uses 40-way 5-terminal (40W5T) switches, controlled by 8 bits per switch. All switches have a single set of control states: they do not allow two paths to use the same switch.

The useful path-sets of the four CAs all allow between 1 to 4 concurrent paths between terminals. Even the linear CA of Fig. 8.24(a) does this, allowing for instance 4 concurrent transfers between adjacent ter-

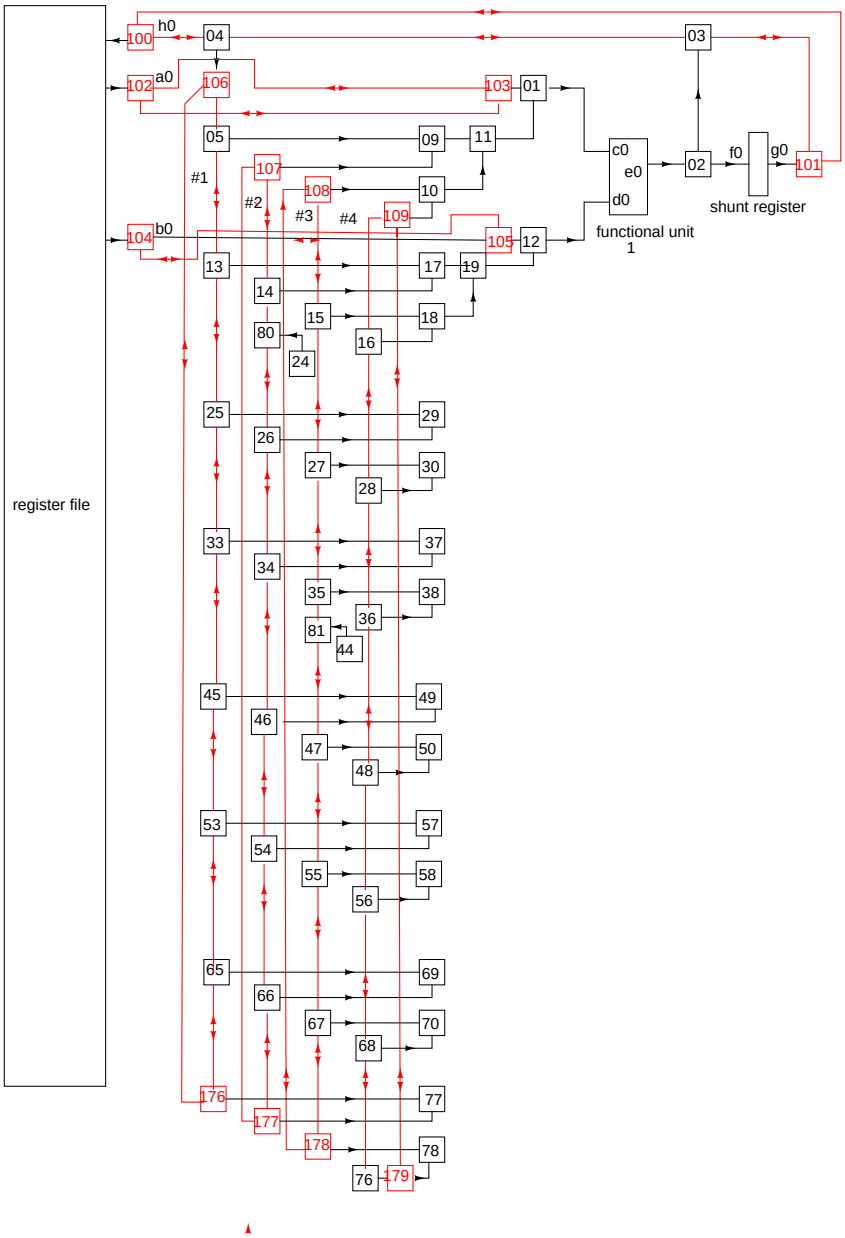


Figure 8.23: Functional unit chaining: survivable network. Out of 16 single-failure fall-back options, 7 are shown in red. Also shown are 14 of the 32 additional switches required. Note that using fall-back rings is not necessarily optimal.

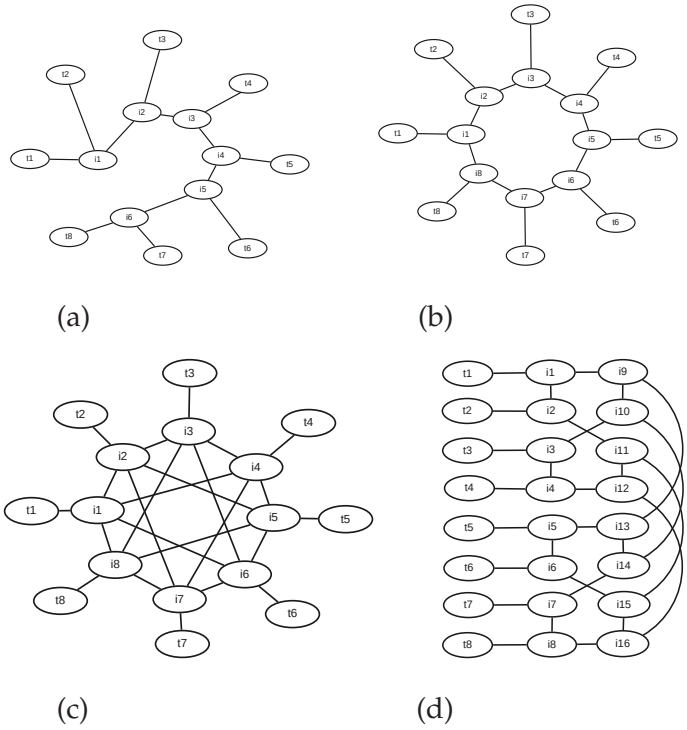


Figure 8.24: Four variable-bandwidth architectures: linear (a), circular (b), K-ring (c), butterfly (d).

minals. Of course, in a linear CA, any end-to-end transfer blocks out all other transfers. The number of useful paths for each CA is rep. 28, 56, 61 and 60. What interests us is the capacity of the topologies to combine the useful paths with reasonable ISG and good UEE.

Table 8.6 shows schedule-neutral ISG and UEE for each CA, together with the number of bits required to control it, derived from the number of concurrent path-sets. The mean unused path length, thus ISG, is lower for the linear and circular topologies. They are simpler and have fewer hops (2.5 resp 4.0, in schedule-neutral conditions) than the K-ring and the butterfly. The ISG for K-ring and butterfly is better, but they also have more hops in the mean (3.5, 4.9 respectively). In all cases, ISG is at or over 50%. This is excellent, but leaves still room for improvement. Thus power-aware placement still makes sense. The UEE is 44% and over, and larger for the more complex CAs. The control bitwidth (10-13 bits) is small enough to find a place in 32-bit and larger

Table 8.6: ISG, UEE and control bitwidth for 4 variable-bandwidth architectures.

	linear	circular	K-ring	butterfly
ISG	58%	50%	78%	75%
UEE	44%	54%	73%	80%
control bits	10	11	13	10

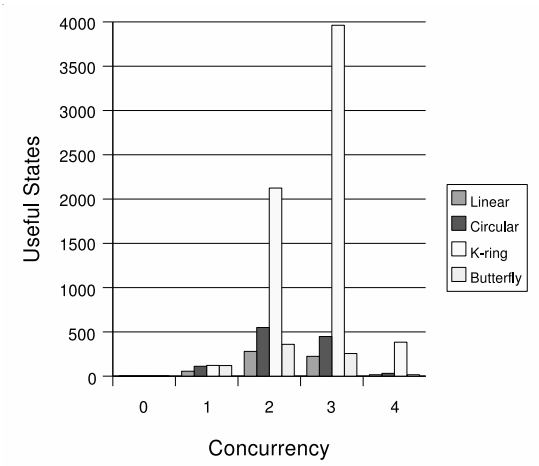


Figure 8.25: Variable-bandwidth CAs: histogram of concurrency.

ISAs.

Comparing the CAs, it is instructive to watch the histogram of the sizes of the concurrent path-sets, shown in Fig. 8.25. Linear, circular and butterfly CAs fail to offer many concurrent path-sets of concurrency number, 2 or 3 in comparison to the K-ring. In fact, the K-ring can at any time route 3 transfers for any combination of terminals, making it far superior to the others in this respect, while it has still a good UEE. The useful-state set of the K-ring can be reduced by excluding path-sets with concurrency number 4, and some with concurrency number 3, that needlessly duplicate combinations of transfers. This does not, as it happens, bring the control bitwidth below 13.

Of the 4 alternatives, the K-ring is superior; offering good ISG, UEE and much concurrency to the benefit of the scheduler. The butterfly does not offer enough concurrency to be of use for in terminal arrangement.

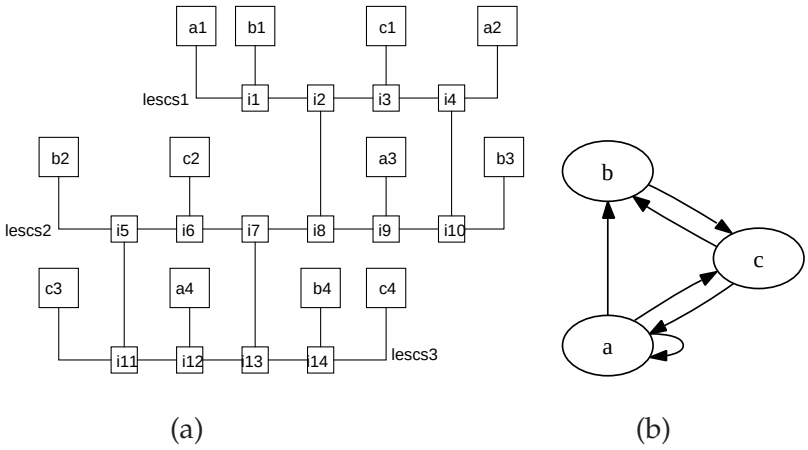


Figure 8.26: Triple linear sectioned CA: topology (a) and terminal arrangement (b), by terminal class.

8.2.3 Broadcasting Arrangements

CA with triple linear topology and narrowcasting Useful-state analysis is applicable to terminal arrangements that feature broad- or narrowcasting. (Narrowcasting has transfers from one source to multiple, but not all, terminals of a class). To show this, we analyze a CA with a network of 12 terminals divided in 3 terminal classes *a*, *b* and *c*. The network contains 16 switches. The network contains 3 linear subnetworks, shown in Fig. 8.26(a). The terminal arrangement is depicted per class in figure Fig. 8.26(b). It imposes bidirectional transfers between terminals of class *a* and *c*, and between class *b* and *c*, and unidirectional transfers from terminals of class *a* to those of class *b*. Moreover, narrowcasting is required within class *a*.

The switches are all 9-way 3-terminal (9W3T) switches, which allow broadcasting and require 4 bits of control. The useful path tree set contains 44 simple paths and 12 narrowcast path trees. Of the alternative paths and trees, only the shortest ones are kept in the useful path-set, to save on resources. A useful-state analysis yields 5087 useful states and a UEE of 80%. Control bitwidth is 13 bits, which makes this network controllable from a program. Without USE, we would have needed 64 bits. The requirement for narrowcasting makes the analysis computationally heavier, but does not change UEE greatly. Without narrowcasting, we would have used 6W3T switches, yielding 777 useful states and

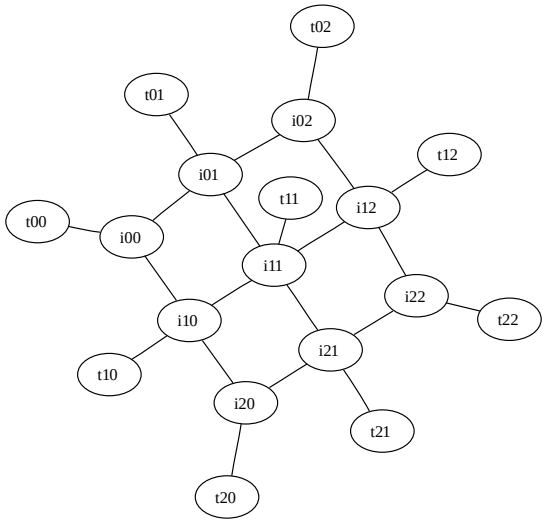


Figure 8.27: Topology of a 4-connected 3x3 grid.

a UEE of 79%, This requires 10 control bits, instead of 48 without USE.

8.2.4 Grid Architecture

In Fig. 8.27 we show a frequently used interconnect topology: the 4-connected grid. Such grids can be found in FPGAs and ASICs connecting logical blocks of various sizes. These architectures are prime candidates for EESC, but are often implemented by means of NoCs, with large overhead. Over a small distance this is overkill. Terminal arrangements can be all-to-all, or else terminals can be of different type. Any CA with grid topology has many alternate paths that may all be useful, offering thus much opportunity for concurrency. On the down side, the combinatorics of this might lead to computational difficulties.

We want to investigate how good our algorithms presently are, and whether they are able to handle CAs of this sort. We choose a typical, but certainly not the easiest terminal arrangement: an all-to-all arrangement, where interconnects are made between any pair of terminals. We analyze only the maximal independent path-sets, and do not try to count or list every single network state. The switches in our example are 24W4T types. Path-finding in the 3x3 grid of Fig. 8.27 works well: all paths are easily found by each of both our path-finding algorithms. It is established within a second, using DIS, that the concurrency num-

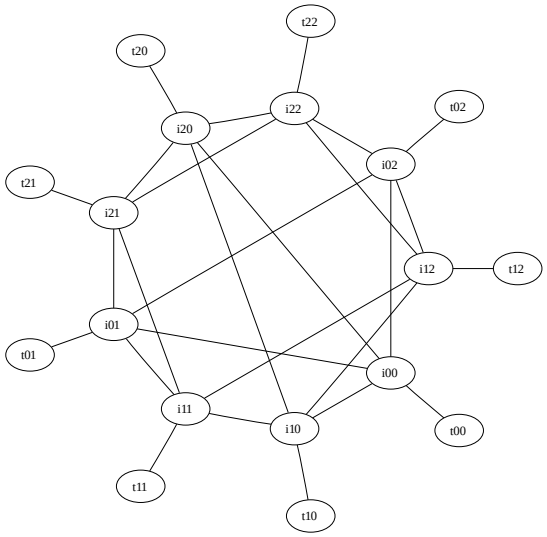


Figure 8.28: Topology of a 4-connected 3x3 torus.

ber is 4. The PAG is established fast (in 3 seconds) and the undirected maximal and complete path-set lookup tables are found in 0.6 and 0.35 seconds, respectively. 502 maximal independent undirected path-sets are found, and 1116 undirected independent path-sets. Given the combinatorics of directionality, the complete undirected path-set tables will turn out to be of impressive size.

Extending the 3x3 grid topology to a 3x3 torus, depicted in Fig. 8.28 we experience the combinatorial power of such topologies. Path-finding algorithms cannot to find all paths, unless we limit the hop count to 6 hops. Since tori are meant to exploit paths with few hops it makes sense to do so. With this limitation, useful paths are found easily. Producing the PAG, however, takes 186 seconds. It must be said that the PAG-making program is not yet optimized and could eventually perform better. The resulting PAG is a large (it has 576 nodes and 158409 edges) and dense graph (it has edge density 0.477) that we can impossibly handle with our algorithms, even optimized. In this huge graph, the DIS algorithm still finds one maximal independent set, and fast: in 16.6 seconds. But we need many more than one set. To control tori featuring EESC, stronger algorithms are required than ours.

If we extend only the size of the 3x3 grid, the number of combinations that generate paths remain manageable. If a reasonable hop limit (8 hops) is used, finding all paths in a 4x4 grid with all-2-all TA (of

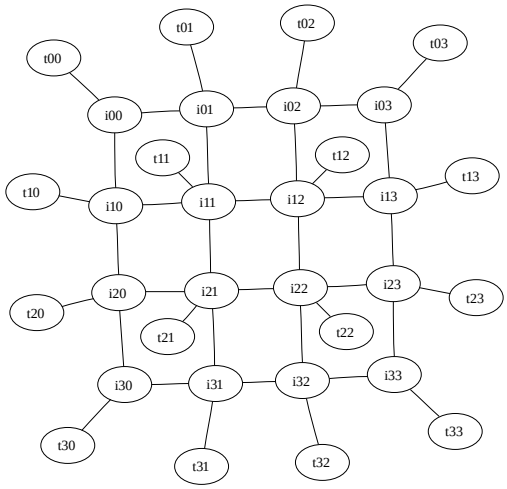


Figure 8.29: Topology of a 4-connected 4x4 grid.

which the network is shown in Fig. 8.29), is possible with the optimized FAR algorithm. We obtain 1432 useful paths; The PAG construction tool must still be optimized; it is now running at its limits. Anyway, the size of this PAG (1432 vertices) is again too large for the present state of optimization of our USA algorithms.

We find that for large grids, featuring EESC, to be controlled from a program, we need a better program for USA. The useful-paths set is simply too large for our approach.

Chapter 9

Conclusions

The conclusion is the place where you stopped thinking.

Near the end of this work, we recapitulate the key themes: the comparison of EESC, from the control viewpoint, with competing methods, the advantage of programmed control, the principles of optimization in the control plane, the theory of useful-state encoding, the design pattern, the methods and algorithms developed, and the experience gathered from use cases.

We will suggest future work, located in the study of the far end of scalability (for extensive intra-tile networks), the development of topology-aware processor architectures, the extension of USA to serve for buffered communication with "store-and-forward", and SoCs that can survive interconnect degradation.

THIS thesis has shown that programmed control over EESC is well possible and can be used for large and complex intra-tile communication networks. We demonstrated a design framework for control planes using statistical analysis. Alternately, concentrating on topological features we obtain, early in the design process, a provisional idea of the sectioning gain that will be achieved, then design the control plane in an optimal way, and ultimately confirm the feasibility of control of EESC. With the control plane implemented in this fashion, the balance of costs remains favorable over a large range of sizes and topologies.

Our method is suited to the present advance into the deep submicron domain: we can exploit opportunities for heterogeneity and parallelism, address the power consumption problem for SoC interconnects at various architectural levels, contribute to solving the problem

of degradation of interconnect reliability, and can control topology-aware processors or terminal arrangements that feature narrow- or broadcasting.

With respect to the future, we must discrimate between *(i)* sectioned communication, *(ii)* control of EESC, and *(iii)* useful-state encoding. The principle of sectioned communication is universal whenever communication links represent costs; its implementation depends on technology. CMOS switches and wire links may be replaced, maybe by nanotube circuits, but sectioning will remain an advantage. Programmed control of sectioned communication is suitable for simple processors. Our design framework is adaptable to others. The trade-offs may change; static and design-time scheduling keep their advantages. Useful-state encoding has broad application, not linked to any technology. It is suited for some future technologies like, for instance, optical-only switching, which precludes "store-and-forward" (there are no optical registers).

9.1 Resource-efficient Control of On-Chip Communication

The reader will find here an overview of the salient points of the thesis. Taken together, these constitute our design framework for the EESC control plane.

EESC occupies a niche in low-power on-chip communication. It concerns medium-distance, synchronous communication within a tile, under control of a program. A NoC, in comparison, has superior network scalability because of "store-and-forward" operation, but incurs an unnecessary overhead, in size and latency from run time arbitration and scheduling. Alternatives, like arbitrated buses, VLSI crossbars and synthesized communication logic, suffer either from unnecessary overhead for resource contention or inferior scalability with the number of terminals involved in communication.

Programmed control is but an application of Wilkes' 1951 proposition: "To efficiently control hardware, design a programming language to execute control decisions using flow control from programs, instead of just more hardware. This avoids decisions having to be taken by dedicated circuitry, and efficiently re-uses circuitry able to interpret the control flow of the program."

Communication processor The communication processor (CP) is a paradigm used to unravel different viewpoints during design, compilation and operation. It assumes static scheduling, and frequent reconfiguration. The first assumption is for simplicity of processor architecture; it still covers many embedded VLIWs and some research processors. The second assumption ensures that efficiency of control plane design really matters, since the volume of control information is high. The latter assumption implies that we should not include registering (buffering) elements in our control plane, and thus must not study differential styles of control. This simplifies design options, and is justified by a use case showing that frequent memory accesses do not offset energy losses from transport of control information over wires.

The principle of control plane optimization is the reduction of the width of the control path at all stages of control: i.e. fetching of communication instructions, path decoding, control processing and distribution of control information over the surface of the chip.

The communication processor leads to a recognition that, when establishing the communication architecture, the SoC's instruction set architecture is paramount, and to the discrimination between implicit and explicit path (or transfer) specification.

Implicit path specification, which uses the instruction set architecture optimally, avoids incurring costs of fetching control information for the communication network. Explicit path-set specification is inefficient because path specification is already implied by the instruction set architecture, and because the compiler must be modified for the purpose of EESC control alone. It can be used when the controlled domain is not programmable, i.e. when the CP is a "stand-alone" sequencer and not integrated in a SoC.

Transfer-, communication- and topology-awareness are terms we use when studying the properties of an instruction set architecture with respect to programmed control of EESC. Topology-awareness is a special case of communication-awareness, and should be distinguished from simple transfer-awareness, which is trivial: all instruction set architectures initiate transfers. Non-trivial communication instructions pertain to the communication architecture itself. Topology-awareness means that the instruction set architecture provides the means of selecting individual path-sets for a transfer through the topology. In contrast to high-performance computing, topolopy-aware SoCs do not really exist yet: It is the purpose of advanced network design to design them,

and of the concept of useful-state encoding to control them efficiently. There are large gaps in topology design for SoC, but they fall outside the scope of this work.

Useful-state encoding Useful-state analysis (USA) reduces the width of the control plane significantly. Useful-state encoding (USE) is of minimum-redundancy, making the control plane optimal in terms of size. USA can be performed for any topology. The path-set lookup table PSLT, resulting from USA, is the interface between communication architecture (CA) and compiler. The concept of USA supports topology-aware processing, adaptive path decoding, and broadcasting.

Useful-encoding efficiency (UEE) is the measure of the success of USA in keeping the control plane small. It is a property of the topology of the communication system, but also of the useful transfer set, and is derived from the SoC's instruction set architecture. A certain amount of sectioning gain is intrinsic to the communication architecture. It can be calculated, independent from any application, in a straightforward way. We call it the intrinsic sectioning gain (ISG) in schedule-neutral conditions. Data-plane design techniques, like power-aware placement and sub-word selection, can improve on this value. In our experience, segmentation gain does not diminish, but rather increase, with increasing topological complexity.

Design pattern The concept of a CP leads to identification of the stages that must be present in any control plane: scheduling, fetching, splitting, transcoding, transporting and decoding.

Each stage has its own costs. In our reference design, fetch costs are minimized by implicit path specification. Area and cost of the path decoder are limited by useful-state encoding, and losses from control energy transport are lower than the energy saved by sectioning, as long as the number of sectioning switches is not too large. Combined topological and statistical study of the application would be needed to determine this number, but we showed to increase with better ISG and UEE. Decoding costs in the sectioning switches, though small in in our use cases, could ultimately scale to become dominant. of this condition can be delayed by by clustering the switches by subnetwork, or by choosing the numerical values of control codes after topological and statistical analysis.

The problem of interconnect reliability is addressed by provid-

ing an adaptive path decoder, driven from a suitable instruction set architecture. The design of such a path decoder is still guided by USA. The computational complexity of USA for survivable networks is known to be severe. The adaptive path decoder is analyzed at design time, when the cost of using massively parallel computing resources is may be justified.

Analysis and design Methods of analysis and design for EESC control planes are seen as *(i)* statistical, where the data of the application, the geometry, and the technology node are taken into account, or *(ii)* as topological, where abstraction is made of these, but we concentrate more on the opportunity of optimization offered by USE. Another approach, *(iii)*, combined statistical/topological, still makes abstraction of the aforementioned factors but takes transition frequencies between useful states in account. This method of analysis can answer all questions of scalability, but it belongs to the future work. We have not yet done such analysis. In a possible approach, a simple graph description is combined in a System C simulator with generic models for switches and the network, and the results of switch decoder synthesis. The simulator would allow to build testbenches for validation of the design, but also allow to inject statistical properties of scheduling into the model.

Algorithms Algorithms for Useful-state Analysis perform sufficiently well at the level of complexity considered in this work. With many more terminals and resources, we might run into problems to find all routes in a communication architecture, or to determine the useful states. For the first problem, the body of mathematical knowledge is limited. For the second, would is possible to advance performance by a wider algorithmic search. For both applications, massive computing resources could employed, if the need arises. This is justified by the benefits of EESC.

Three algorithms for USA and their implementations were investigated: (maximal independent path-set, independent path-set and path-powerset algorithm with Dharwadker's independent set algorithm). MIPS scales best and can be parallelized; PPS with DIS is suitable for massive parallelism. Alternate approaches to USA, like linear programming/integer (linear) programming techniques, can either expand the applicability of USA, or open new roads towards (relative) efficiency. We must recognize however that ultimately, combinatorial

complexity will always pose a challenge.

9.2 Summary of Future Work

Future work includes determining the weight of the transcoding and decoding losses mentioned in Section 6.3 based on combined statistical and topological analysis, using not only the frequencies of occurrence of the useful states, but also the transition frequencies between them. This would enable us to quantify these losses, and discern the upper limit of scalability. The crucial point here is how to characterize the behavior of the scheduler without entering into detailed simulation or profiling of the application.

A particular approach, maybe undervalued in our study, can alleviate computational complexity. It was mentioned in Section 4.2 that Useful-state Analysis can be performed by combining the results for subnetworks with known path-set lookup tables, upon which USA was thus already performed. Lacking previous analyses, and not having developed tools for this combination, we did not explore the avenue. It is reasonable, however, to expect that, this "divide-and-conquer" approach can yield serious dividends and optimizations.

A large body of knowledge of advanced topology design exists in the field of high-performance computing; it is not known how well it fits into the context of SoC. Also, at this moment, the requirements for survivability in the face of reliability degradation are not yet well-formulated. As this knowledge becomes available, USA can be employed to study the practicality of control in these advanced networks which have interesting properties.

We have not had the opportunity to actually design a topology-aware embedded processor, partly because useful-state encoding became not available early enough. With USA, it should now be possible to design SoCs with communication architectures that are both power-efficient and survivable.

An interesting theoretical prospect is to extend the validity of USA to domains that extend over multiple clock periods, by incorporating buffering in the switches. This would extend the niche that EESC occupies to inter-tile communication on SoC, and enable the application of sophisticated communication architectures in CMT/CMP applications.

Dynamically programmed embedded processors IMEC [38, 99, 100, 101, 122] and other research institutions are planning development of a class of embedded processors called *dynamically programmed*, in response to a increased level of dynamism in applications and the combined requirements of process variability [104] and aging effects [50, 93]. Some of these new platforms are disruptive in the sense that they are not conventional and do not follow the communication processor paradigm. Nevertheless many results of our work are potentially applicable to this new type of processor, including the importance of a topology-awareness, USA, the benefits of useful-state encoding and the trade-off between implicit and explicit path specification.

Appendix A

Design Support Tools

This appendix describes a convention to allowing describing and visualizing network topologies, and usage forms for tools used in support of design, analysis, simulation, and synthesis.

SINCE they handle both directed and undirected graphs and multi-graphs, allow visualization of large graphs (`dotty(1)`, tulip), and feature in extensive graph algorithm libraries (`perl`'s `Perl::Graph`[1] and the Boost Graph Library (BGL), the `dot(1)` graph description language and the GraphViz [40] visualization system were chosen for all graphs involved in USA.

A.1 Graph Description and Visualization

Using a graph description as a network description, some conventions must be respected: *(i)* A graph defines vertices (terminals and internal resources, e.g. switches) and edges (wires, 'nets' in the language of synthesis). It does not attribute an order to the edges. If the edge is seen as a net, the port number of the terminal or resource that the net is attached to is not implied by the graph.

A terminal has only one port, so for terminals there is no problem. For resources, we must circumvent the ambiguity. The convention is, then, for resources, that nets are attached to the ports of the resource in the alphabetical order of the names of the corresponding neighbor

[1] Perl tools require `Perl::Graph` version 0.69 or later.

vertices, i.e. of the resources that the edges are incident on. *(ii)* Terminals can be distinguished from resources by their vertex degree, but not all support tool programs do this, as it may spoil their generality. The convention is to start the name of resource vertices with `i` (for *internal*) or `sw` (for *switch*), and use *t* or a terminal class indication for terminals. Given these conventions and the types of resource, a graph specification becomes a network hardware description.

A.2 Finding an All-Paths Set: `far`, `fdr`, and `fdr2`

Two ways exist to establish the all-paths set: FAR and Grover's FDR algorithm. `far`'s form of usage is:

The FAR algorithm

```
far: usage: far [-Dhv] [-H maxhopct] [-i internal-prefix] graph.dot
options: -D:  directed graph
  -h:  print help
  -H maxhopct:  maximum hop count
  -i internal-prefix:  prefix to recognize a vertex as internal
  -v:  print revision id
infile: .dot (GraphViz) graph representation
```

`far` is used to generate all short distinct routes in a network topology. Use it only with small hop counts. It consumes large amounts of time and memory when the hop count gets too large.

Grover's FDR algorithm

```
fdr: usage: fdr [-dR] [-s seed] [-h maxhopct] [-l logtree]
  [-r rtelimit] -f from -t to infile
options: -d: debug
         -R: not random; always choose first available
             ('leftmost') edge
         -s: seed for randomizer
         -h maxhopct: maximum hop count
         -l logtree: file to log search path in
         -r rtelimit: limits number of routes to print
         -f from: source vertex
         -t to: target vertex
infile: .dot (GraphViz) graph representation
```

`fdr` is used to explore the behavior of Grover's find-direct-routes algorithm. It can find longer routes, but only a few at a time, and it has

a tendency to block on long routes in irregular networks. Typically, use it a few times like this:

```
$ fdr -h 12 -r 3 -f t1 -t t5 topology.dot
```

Note the number of different routes it regularly produces, like for instance two or three at a time. In that case, use 3 as the route limit for `fdr2`. `fdr2`'s form of usage is:

```
fdr2: usage: fdr2 [-Uv] [-c ctlimit] [-h maxhops]
  [-k known-routes] [-r rtelimit] [-R totalrtelimit]
  -f from -t to infile
options: -U: update known-routes-table
         -v: verbose (recommended)
         -h maxhops: maximum hop count
         -c ctlimit: limits number of runs of fdr
         -r rtelimit: limits number of routes per run of fdr
         -k known-routes-table: routes table to use
         -R totalrtelimit: limits total number of routes
         -f from: source vertex
         -t to: target vertex
infile: .dot (GraphViz) graph representation
```

`fdr2` can be used to fill up (update) a known-routes table until no more new routes are found. This runs `fdr` repetitively. Each of the runs can be interrupted with *CNTL*-C if it no longer produce new routes. It is recommended to use the `-v` option to observe what is happening, like in this example:

```
$fdr2 -vUk routes.txt -h 12 -r 3 -R 100 -f t1
   -t t5 topology.dot
```

For small regular graphs, `fdr2` can sometimes establish all paths without user intervention. A small script for this is included in Section A.5.

A.3 Making a Path Allocation Graph: `mkpag`

`Mkpag` still works well for most communication architectures, but is in need of optimization.

NAME

```
    mkpag - make a path allocation graph (PAG) or path resource
    allocation graph (PRAG)

SYNOPSIS
    mkpag [-ADhLmNrv] [-n name] nw.dot [paths-file] [path-trees ...]

OPTIONS
    A PAG/PRAG is constructed from a network graph file "nw.dot", plus
    optionally a set of a useful paths, listed in the paths-file. When
    broadcasting is involved, a set of broadcast path trees may also
    be specified. A PAG/PRAG is always an undirected graph. The paths
    it refers to may be undirected or directed.

    In the most common case, The network graph file is undirected, and
    the useful paths given are also undirected. This needs no special
    options. A PAG/PRAG for undirected paths is constructed. If
    broadcast path trees are additionally given, these are directed
    path trees. A PAG or PRAG for directed paths should then be
    constructed and this requires the -D option. On the other hand, if
    the network graph is directed, the paths-file must contain
    directed paths. (Broadcast path trees may additionally be given.)
    A PAG or PRAG for directed paths should then be constructed; this
    requires the -A option.

    -A      Make a PAG/PRAG from a directed network graph, referring
            to directed paths. The paths in the paths-file are
            required to be directed, and path-trees are always
            directed. (The resulting PRAG/PAG still is an undirected
            graph. The PSLT made from the PAG will be called a
            directed pathset lookup table, since it contains codes for
            directed paths.)

    -D      Make a PAG/PRAG referring to directed paths. The paths in
            the paths-file are required to be undirected, and
            path-trees are always directed. The network graph must be
            undirected, i.e. bi-directional. (The resulting PRAG/PAG
            still is an undirected graph. The PSLT made from the PAG
            will have codes for directed paths. This is
            computationally much heavier but must be done if
            undirected and directed paths are mixed, since otherwise
            it is difficult to count useful states.)

    -h      Print a help message.

    -L      Don't label edges.

    -m      Prints the manual page.

    -n "name"
```

```
            Name to give to graph.

    -N      Include a null path. This makes sense for PRAGs.

    -r      Make a PRAG, not a PAG.

    -v      Be verbose.

DESCRIPTION
    mkpag reads a network graph file, optionally also a paths file and
    a set of path trees. It writes the PAG as a dot graph file, to
    standard output.

    The path file consists of lines containing paths, with the path
    components separated by colons. These paths are undirected, unless
    the -A option is given. Then they are interpreted as directed, and
    the components may be separated by arrows ('->').

    Path trees are directed dot graph files, that may represent either
    additional (directed) paths or broadcast path trees.

    If no paths file is given, mkpag will assume a simple terminal
    arrangement with two terminal sets, and construct the shortest
    paths, using Floyd-Warshall, between the terminal pairs of the
    network. A terminal is any node whose name does not start with "i"
    or "sw" (for 'internal', resp. 'switch'). This is OK for acyclic
    networks, with two terminal classes. In other cases, mileage may
    vary.

Terminal and resource nodes
    The nodes of the NWG are either terminals (nodes of degree 1) or
    resources (all others). Their name must have the form 'cn', where
    'c' is the terminal class for a terminal or 'i' or 'sw' for a
    resource, and 'n' is an integer.

Paths
    For the network graph to be a host graph of the PAG/PRAG, all
    paths must be paths of the network graph. For useful state
    analysis, the paths must be resource-disjoint.

PAG
    A PAG is a simple graph: its vertices are paths from the paths
    set. Its edges represent sets of resources that are common to the
    paths.

PRAG
    A PRAG is a bi-partite graph: its vertices are partioned into
    resources and paths. It is the bi-partite (Koenig) representation
    of the resource allocation hypergraph (RAH). The RAH is the dual
```

```
    hypergraph of the PAG. The RAH's vertices are resources; its
    hyperedges are paths incident to each resource they contain.

RETURN VALUE
    0 if succesful and non-zero on failure.

SEE ALSO
    mkram(1), mkta(1).

AUTHOR
    Kris Heyrman <kris.heyrman@imec.be>
```

A.4 Making a Path-set Lookup Table: `mkpslt5`

`mkpslt5` uses auxiliary programs for good performance. By default, an IPS (independent path-set) algorithm is used. With the `-6` option, it uses PPS (path powerset). With the `-7` option, it uses MIPS (maximal independent pathset), which is for most problems the fastest. The auxiliary programs can also be used directly, but do not necessarily have all options that `mkpslt5` has.

```
NAME
    mkpslt5 - make a complete path set lookup table (PSLT), from a
    path allocation graph (PAG)

SYNOPSIS
    mkpslt5 [-67dDhmMRv] [-k k] [-T tl] -o output-file pag.dot

OPTIONS
    -6      Use PPS algoritm, not IPS.

    -7      Use MPS algoritm, not IPS.

    -T tl   Calculate mean hop length (MHL) and intrinsic sectioning
            gain (ISG), over all useful states, given this total
            length of the topology in hops. (Not counting the null
            US.) Also prints the number of vertices in the PAG, and
            the amount of state reduction caused by the -R option, if
            any.

    -D      Write a directed PSLT, not an undirected one.

    -d      Don't die if a PAG with mixed undirected/directed paths is
            presented. In this case, better call the resulting PSLT
            'mixed' and recombine it to a directed PSLT by other
```

```
            means.

    -h      Print a help message.

    -k "k"  Include at most "k" paths per set. (I.e. find k-maximal
            independent sets.) Do not use with -6 option.

    -m      Print this manual page.

    -M      Write a maximal PSLT, not a complete one. Can only be
            done together with -7 option.

    -o "output-file"
            Output as output-file. Mandatory option.

    -R      Reduce (disambiguate) the path set lut to a transfer set
            lut. Whenever 2 path sets contain different paths between
            the same sets of terminal pairs, only the path set with
            minimal total hop count is kept.

    -v      Be verbose.

DESCRIPTION
    mkpslt5 writes a PSLT to the output file. This PSLT may be
    directed or undirected (containing directed/undirected paths),
    reduced (to a transfer-set lookup table) or not,

IMPLEMENTATION
    mkpslt5 uses auxiliary programs for higher performance. By
    default, an IPS (independent path-set) algorithm is used. With the
    -6 option, it uses PPS (path powerset). With the -7 option, it
    uses MIPS (maximal independent pathset), which is fastest for most
    problems. The auxiliary programs can also be used directly, but do
    not necessarily have all options that mkpslt5 has.

RETURN VALUE
    0 if succesful and non-zero on failure.

BUGS
    PAGs cannot have edge labels, or the auxiliary program will abort.
    This is actually a feature. They must have been made with
    mkpag(1)'s -L option.

SEE ALSO
    anapslt(1), mkpag(1).

AUTHOR
    Kris Heyrman <kris.heyrman@imec.be>.
```

A.5 Sample Makefile for USA

The user should not expect to analyze a communication architecture automatically, without intervention. It is recommended to work with a `make(1)` and a Makefile, since files produced by USA have some dependencies. Below is an sample Makefile that can be employed, given a topology, to find an all-paths set, determine the useful paths and analyze a communication architecture. The variable `STEM` should be set to the common stemname of the files to be produced by the Makefile.

```
# Makefile for useful state analysis
# remember that lines with rules start with tabs

# all-paths files, path allocation graph, complete path set lut
all: topo.grover.allpaths \
     topo.far.allpaths \
     topo.pag.dot \
     topo.Cpslt \

STEM = topo # stemname for all files

# default rules

# find useful paths (FDR algorithm)
%.grover.allpaths: %.dot
        >$*.grover.allpaths
        for FROM in f1 f2 f3 f4; do \
          for TO in m1 m2 m3 m4; do \
            fdr -f $$FROM -t $$TO -h 9 -r2 $*.dot \
              >>$*.grover.allpaths; \
          done \
        done
        sort $*.grover.allpaths >$*.grover.sorted.allpaths
        mv $*.grover.sorted.allpaths $*.grover.allpaths

# find useful paths (FAR algorithm)
# FAR finds all-to-all paths, so must be filtered
%.far.allpaths: %.dot
        far5 -ii -H9 $*.dot >$*.far.allpaths
        sort $*.far.allpaths >$*.far.sorted.allpaths
        mv $*.far.sorted.allpaths $*.far.allpaths
        sort $*.far.allpaths \
          | grep -v "^f.*f.$$" \
          | grep -v "^m.*m.$$" >$*.far.selected.allpaths
        mv $*.far.selected.allpaths $*.far.allpaths

# make a path allocation graph
# the usefulpaths file is either of the 2 above
```

```
# allpaths files, (if all paths are useful)
# or your own useful paths file, edited from either
%.pag.dot: %.dot %.usefulpaths
        mkpag -n $*.pag $*.dot $*.usefulpaths \
          >$*.pag.dot

# make a reduced directed complete pathset lookup table
%.Cpslt: %.pag.dot
        mkpslt -DR -o $*.Cpslt $*.pag.dot

# below enter specific rules
```

Appendix B

Components of the Control Plane

This appendix contains circuit diagrams and HDL sample code of components in the EESC control plane: the sectioning switch, the network and the path decoder.

We include the System C RTL model for a 6W3T switch and for the network used in our simulator, a future scheme for network exploration, and an example circuit diagram of a path decoder.

B.1 6W3T Sectioning Switch

6W3T sectioning switches belong to both data and control plane. We define here the interface between the switch and the data plane. We show implementations of the 6W3T sectioning switch for LP, the low-leakage option of semiconductor foundries, where dynamic losses dominate over leakage, and for HP process technologies where static losses dominate.

Circuit diagrams Rabaey [97] reviews tri-state circuits for CMOS-technologies with dissipation that is mainly dynamic, discussing two different versions of a CMOS tri-state buffer. Fig. B.1 shows his preferred choice to drive large capacitances. Tri-state circuits do not themselves limit power consumption by leakage; EESC does this indirectly by reducing the size of the buffers.

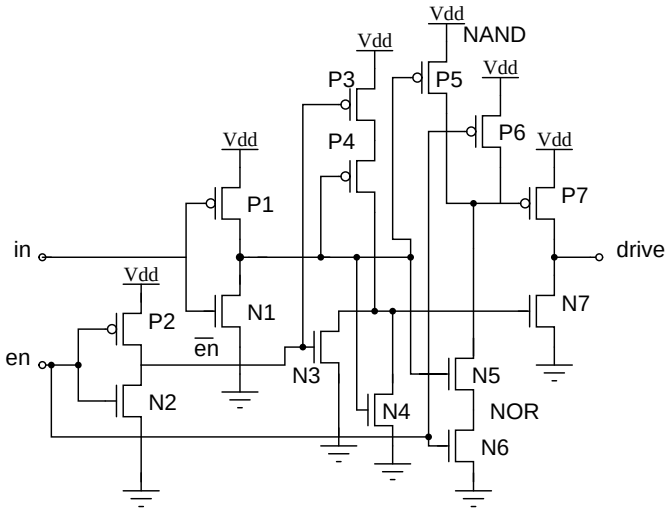


Figure B.1: Circuit diagram for one of the three output stages in an EESC tri-state switch.

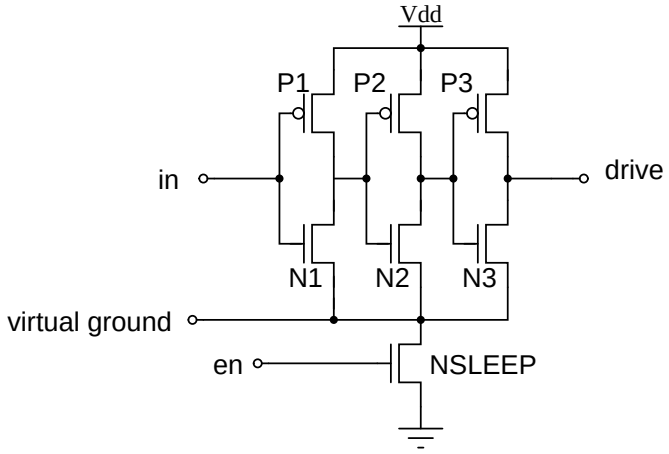


Figure B.2: Circuit diagram for the output stage of a power-gating switch.

Power-gating circuits, used against dissipation that is mainly static, reduces leakage by itself. In a possible implementation, a "power gate for wires" from Pedram [95, 72], shown in Fig. B.2, only one NMOS transistor is used to switch off all buffers at once. An NMOS has lower on-resistance than PMOS. Two transistors could be used, but using one is practical and saves area. This is termed *sleep-transistor sharing*.

Power gating has some drawbacks: it requires a modified CMOS technology to support both low threshold-voltage devices for logic and high threshold-voltage devices for sleep devices, decreases the voltage swing, and thus the DC noise margin, and does not scale well with supply voltage. Moreover, sizing sleep transistors is not a trivial task.

Still according to Pedram, a trade-off exists between dynamic power consumption by the sleep transistor and static power-saving in a circuit like Fig. B.2. A large sleep transistor decreases the high-to-low transition delay in the circuit, but has more area overhead and consumes extra dynamic power when turning on and off. Further, a minimum time may exist below which power saving is not effective. Therefore we do not want to argue that Fig. B.2 offers a definitive solution, only that the topic has open issues and is in need of future work. This proposed circuitry for DSM technologies must still be run past technology experts at IMEC.

System C model The class `sw6w3t`, listed in listings B.3-B.6, is a RTL model for a physical switch. We observe the class constructor, the simulation method `prc()`, and methods for loading the switch lookup table (`load_lut()`) and setting all ports to high-impedance at the start of simulation (`async_reset()`). The model defines the 7 states of a switch and features control inputs and sub-word selection, a reset input (for simulation only) and 3 bidirectional in/out ports.

For versatility during synthesis and simulation, a switch is implemented as a template class, that can be instantiated for different word width `W`, control plane width `WCP`, sub-word width `SLICE`, and sub-word selection control width `WSW`. The model can be used for a single switch, but also for a word-wide switch, with common decoder for all wires.

By default, the switch object is constructed for control plane width `WCP=3`; this is our 'default switch' of Fig 3.2. In communication architectures controlled by USE, the width of the control plane is decided from USA and the default switch lookup tables are overwritten by the proper lookup table for an individual switch during construction of the network.

The model can be expanded to a switch with more ways and terminals (an "xWyT" switch); it is not synthesizable, but writing an equivalent synthesizable model in any HDL is a straightforward task.

B.2 Network Synthesis

Class `network`, is templated with the same template parameters as `sw6w3t`. The interface definition is shown in listing B.1. The `network` class has a constructor from a simple graph description file and contains a pointer to a vector of switch lookup tables. Using `network`'s constructor, a network, including switches and switch lookup tables can be instantiated from only a graph description file and a lookup table initialization file. The code for this is shown in listing B.2. In the example, the network is 32-bits wide, has a word width `W=32`, 5 bits for USE (the control path width `WCP=5`), byte-sized subwords and 4 bits for sub-word selection (in order to individually select 4 bytes in the word).

Validation and Synthesis of Communication Architectures Using the switch and network model, future work will allow validation of USA and synthesis of the communication architecture, including the switch lookup tables and control wire network, given a GraphViz network description. With suitable switch models, the method work for other switch types than 6W3T. This opens up interesting avenues for the exploration of different communication architectures with different topologies and switches.

B.3 Path Decoder

The GSM speech encoder with loop buffers (the use case in Section 8.1.2) illustrates a specific path decoder, designed for a single-path linear bus CA. Since the bus network is symmetric around its center, we only need a path decoder for half a bus network. (For a full bus network, the path decoder is shared between the two halves.) For a scratchpad memory with a load/store unit and 2x4=8 memories, the path decoder consists of 68 NAND gates and 1 latch. It scales better than quadratically in size with number of modules ($N = 8, 16 \ldots$). If more bus networks ($M = 1, 2, 4 \ldots$) exist, the path decoder can be multiplexed per network; it is then of constant size with an increasing number of the buses. If the number of buses is not too large, a form of bus pipelining can be employed. If not pipelined, the inherent latency of a multiplexed path decoder is O(M) with the number of buses.

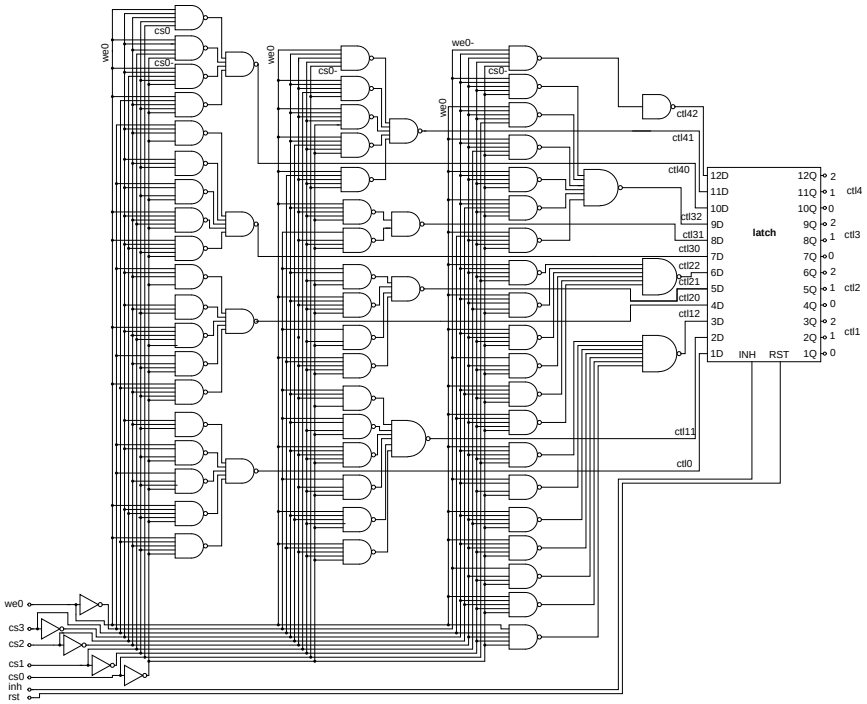


Figure B.3: Path decoder, for half a LSB of order 8.

```
template < int W = 1, // (word) bitwidth of network
           int WCTL = 3, // width of ctl network
           int SLICE = 1, // slice bitwidth for sw
              select
           int WSW = 0, // width of sw select ctl
           template <int, int, int, int>
              class SWITCH = sw6w3t > // switch
                 type:
struct nw : sc_module {
  // in/output signals
  sc_in < sc_uint < WCTL > > ctl; // control bits
  sc_in < sc_uint < WSW > > sws; // subword select
     mask
  sc_in < bool > rst; // reset; for simulation only
  sc_inout_rv < W > t0; // terminals
     /* ... */          // more terminals
  sc_inout_rv < W > t8; // last terminal

  // data members
    /* ... */

  // constructor
  nw(sc_module_name name, // module name
     std::string filename, // name of GraphViz file
     std::vector <
       std::map < sc_uint < WCTL >, state >
     >* tablesp = 0); // points to vector of luts

  // load non-default-type switch control lookup
     tables
  virtual void
  load_luts(std::vector <
              std::map < sc_uint < WCTL >, state >
            >* tablesp);
};
```

Listing B.1: Class interface definition for a network.

```
// a vector of lookup tables
vector < map < sc_uint <5>, sc_uint<3> > > luts("
   luts.ini");

// the network
nw < 32, 5, 8, 4 > dut("dut", string("graph.dot"));

// load the lookup tables
dut.load_luts(&luts);
```

Listing B.2: Constructing the switch lookup tables, from a switch lookup description file `luts.ini` and a device under test, i.e. a network, from a graph description `graph.dot`. After construction, the device's lookup tables are loaded.

```
// sw6w3t.h: n-tuple 6-way 3-terminal switch

typedef sc_dt::sc_uint < 3 > state;
const state ctl0to1  = "000"; // 0->1
const state ctl0to2  = "001"; // 0->2
const state ctl1to2  = "010"; // 1->2
const state ctl1to0  = "011"; // 1->0
const state ctl2to1  = "100"; // 2->1
const state ctl2to0  = "101"; // 2->0
const state illegal  = "110";
const state ctldisc  = "111"; // disconnected

template < int W = 1, /* or 32, for instance */
           int WCP = 3,
           int SLICE = 1, /* or 8, for instance */
           int WSW = 0 > /* or 4, for instance */
```

Listing B.3: Class describing a 6W3T switch, part 1.

```
struct sw6w3t : sc_module {
  // in/output signals
  sc_in < sc_uint < WCP > > ctl; // control bits
  sc_in < sc_uint < WSW > > sws; // subword select
     mask
  sc_in < bool > rst; // reset; for simulation only
  sc_inout_rv < W > io0; // first, downstream, left
      ...
  sc_inout_rv < W > io1; // second, drop, middle
     ...
  sc_inout_rv < W > io2; // third, upstream, right
     ...

  // data member: ctl lookup table
  map < sc_uint < WCP > , state > lut;

  typedef sw6w3t SC_CURRENT_USER_MODULE;
  // constructor
  sw6w3t(sc_module_name name,
     map < sc_uint < WCP >, state >* tablep = 0)
  : sc_module(name) {
    SC_METHOD(async_reset); // first
    sensitive << rst;
    SC_METHOD(prc);
    sensitive << ctl << sws << io0 << io1 << io2;

    // test some assumptions about template
       parameters
    if (WSW != 0) assert(W == SLICE * WSW);
    //if (WCP != 3) assert(tablep != 0);

    // initialize lookup table
    if (tablep == 0) // default lut for 6W3T
      for (int i = 0; i < (1<<WCP); i++)
        lut[i] = state(i);
    else
      load_lut(tablep);
  }
```

Listing B.4: Class describing a 6W3T switch, part 2.

```
// systemc method
virtual void prc() {
  // apply lookup table to control value
  state ctl_tmp = lut[ctl.read()];
  if (ctl_tmp == illegal) assert(0);

  // read switch input
  sc_lv <W> i_tmp =
    ctl_tmp == ctl0to1 or ctl_tmp == ctl0to2? io0
       .read():
    ctl_tmp == ctl1to0 or ctl_tmp == ctl1to2? io1
       .read():
    ctl_tmp == ctl2to0 or ctl_tmp == ctl2to1? io2
       .read():
    sc_lv <W> (SC_LOGIC_Z); // disconnected

  // apply subword selection
  for (int i = 0; i < WSW; i++)
    if (sws[i] != 0)
      for (int j = 0; j < SLICE; j++)
        i_tmp[i*SLICE + j] = SC_LOGIC_Z;

  // write output
  if (ctl_tmp == ctl0to1) {
    io1.write(i_tmp); // 0->1
    io2.write(sc_lv <W> (SC_LOGIC_Z));
  } else if (ctl_tmp == ctl0to2) {
    io2.write(i_tmp); // 0->2
    io1.write(sc_lv <W> (SC_LOGIC_Z));
  } else if (ctl_tmp == ctl1to2) {
    io2.write(i_tmp); // 1->2
    io0.write(sc_lv <W> (SC_LOGIC_Z));
  } else if (ctl_tmp == ctl1to0) {
    io0.write(i_tmp); // 1->0
    io2.write(sc_lv <W> (SC_LOGIC_Z));
```

Listing B.5: Class describing a 6W3T switch, part 3.

```
    } else if (ctl_tmp == ctl2to1) {
      io1.write(i_tmp); // 2->1
      io0.write(sc_lv<W>(SC_LOGIC_Z));
    } else if (ctl_tmp == ctl2to0) {
      io0.write(i_tmp); // 2->0
      io1.write(sc_lv<W>(SC_LOGIC_Z));
    } else { // disconnected
      io0.write(i_tmp);
      io1.write(i_tmp);
      io2.write(i_tmp);
    }
  }

  // load non-default-type switch control lookup
     table
  virtual void load_lut(map<sc_uint<WCP>,
     state>* tablep) {
    for (int i = 0; i < (1<<WCP); i++) {
      sc_uint<WCP> tmp(i);
      // all values present in table?
      assert(tablep->find(tmp) != tablep->end());
      lut[tmp] = (*tablep)[tmp];
    }
  }

  // for simulation only
  void async_reset() {
    io0.write(sc_lv<W>(SC_LOGIC_Z));
    io1.write(sc_lv<W>(SC_LOGIC_Z));
    io2.write(sc_lv<W>(SC_LOGIC_Z));
  }
};
```

Listing B.6: Class describing a 6W3T switch, part 4.

Bibliography

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, D. Burger, "Clock rate versus IPC: the end of the road for conventional microarchitectures", Proc. of the 27th International Symposium on Computer Architecture, pp. 248-259, 2000.

[2] R. K. Ahuja, T.L. Magnanti, J.B. Orlin, "Network flows", Prentice Hall, New Jersey, 1993, chap. 17.

[3] R. Allen, K. Kennedy, "Optimizing compilers for modern architectures", Academic Press, San Diego, 2002, chap. 12.

[4] F. Angiolini, P. Meloni, S. Carta, L. Benini, L. Raffo, "Contrasting a NoC and a traditional interconnect fabric with layout awareness", Proc. Design, Automation and Test in Europe (DATE '06), vol. 1, pp. 1-6, Mar 06.

[5] C. M. Aras, Ren C. Luo, D. S. Reeves, "The segmented bus: a dynamically segmentable interprocessor communication network for intelligent mobile robot systems", Proc. of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems, 1992, pp. 309-316, Jul 1992.

[6] F. Arifin, "Application and evaluation of segmented-bus architecture", Master of Science thesis, Royal Institute of Technology, Stockholm, Dec 2004 .

[7] N.D. Arora, Li Song, S.M. Shah, K. Joshi, K. Thumaty, A. Fujimura, L.C. Yeh, Ping Yang, "Interconnect characterization of X architecture diagonal lines for VLSI design", IEEE Trans. on Semiconductor Manufacturing, vol. 18, no. 2, pp. 262-271, May 2005.

[8] "`atlas::bitset::BitSet<n>` Class Template Reference", `http://www-math.mit.edu/dav`

[9] "The ATOMIUM tool suite", http://www.imec.be/design/atomium/

[10] J. Babb, R. Tessier, M. Dahl, S.Z. Hanono, D.M. Hoki, A. Agarwal, "Logic emulation with virtual wires", IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 16, no. 6, pp. 609-626, Jun 1997.

[11] X. BaiQiang, N. Ida , "A dynamically segmented bus architecture", Computers and Electrical Engineering, volume 16, no. 3, pp. 139-158 , 1990.

[12] K. Banerjee, S.J. Souri, P. Kapur, K.C. Saraswat, "3-D ICs: A novel chip design for improving deep-submicrometer interconnect performance and systems-on-chip performance", Proc. IEEE, vol. 89, pt. 5, pp. 602-633, May 2001.

[13] K. Banerjee and A. Mehrotra, "A power-optimal repeater insertion methodology for global interconnects in nanometer designs", IEEE Trans. on Electron Devices, vol. 49, no. 11, pp. 2001-2007, Nov 2002.

[14] K. Banerjee, A. Mehrotra, A. Sangiovanni-Vincentelli, C. Hu, "On thermal effects in deep submicron VLSI interconnects", Proc. Design Automation Conference (DAC 1999), pp. 885-801, 1999.

[15] F. Barat Quesada, "CRISP: A scalable VLIW Processor for Low Power Multimedia Systems", Doctorate thesis, Katholieke Universiteit Leuven, Sep 2005 .

[16] L. Benini et al., "Networks on chips: a new SoC paradigm", IEEE Computer, vol. 35, no. 1, pp. 70-78, Jan 2002.

[17] "Boost C++ libraries", http://www.boost.org/

[18] F. C. Botelho, R. Pagh, N. Ziviani, "Simple and space-efficient minimal perfect hash functions", LNCS-4619, pp. 139-150, Springer-Verlag, 2007.

[19] A. Brinkmann, J.-C. Niemann, I. Hehemann, D. Langen, M. Porrmann, U.Ruckert, "On-chip interconnects for next generation system-on-chips", 15th Annual IEEE International ASIC/SOC Conference, 2002, pp. 211-215, Sep 2002.

[20] G. Broomell, J.R. Heath, "Classification categories and historical development of circuit switching topologies", Computing Surveys, vol. 15, no.2, pp. 95-133, Jun 1983.

[21] D. Buss, "Si technology roadmap for ubiquitous computing, sensing and perception", Intl. Symp. on Circuits and Systems (ISCAS 2007), http://www.iscas2007.org/Assets/busstalk07.pdf.

[22] "CACTI 4.2", http://www.hpl.hp.com/research/cacti/

[23] L. Carloni, A. L. Sangiovanni-Vincentelli, "On-chip communication design: roadblocks and avenues", First IEEE/ACM/IFIP International Conf. on Hardware/Software Codesign and System Synthesis, 2003, pp. 75-76, Oct 2003.

[24] F. Catthoor, "Unified low-power design flow for data-dominated multi-media and telecom applications", Springer, Berlin Heidelberg, 2000.

[25] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, T, Omnès, "Data Access and Storage Management for Embedded Programmable Processors", Kluwer, Dordrecht, 2002.

[26] Vikas Chandra, G. Carpenter, J. Burns, "Dynamically optimized synchronous communication for low power system on chip designs", Proc. 21st International Conference on Computer Design (ICCD 2003), pp. 134-139, Oct 2003.

[27] J.Y. Chen, W.B. Jone, J.S. Wang, H.-I. Lu, T.F. Chen, "Segmented bus design for low-power systems", IEEE Trans. on VLSI, vol. 7, no. 1, Mar 1999.

[28] W. Dally, B. Towles, "Route packets, not wires: on-chip interconnection networks", Design Automation Conference (DAC 2001), pp. 684-689, Jun 2001.

[29] W. J. Dally, B. Towles, "Principles and practices of interconnection networks", Morgan Kaufman, San Francisco, 2004.

[30] "A. Dharwadker - The Independent Set Algorithm", http://www.geocities.com/dharwadker/

[31] J. Duato, S. Yalamanchili, L. Ni, "Interconnection networks, an engineering approach", Morgan Kaufman, San Francisco, 2002.

[32] S. Dutta, K. J. O'Connor, A. Wolfe, "Proc. Ninth Annual IEEE International ASIC Conference and Exhibit, 1996, pp. 45-49", Sep 1996,

[33] U. Eco, "Il nome della rosa", Fabbri-Bompiani, Sonzogno, 1980, chap. "Second day, Night".

[34] D. Eppstein, "Small maximal independent sets and faster exact graph coloring", Journal of Graph Algorithms and Applications, vol. 7, no. 2, pp. 131-140, 2003.

[35] M.J. Flynn, P. Hung, "Microprocessor Design Issues: Thoughts on the Road Ahead", IEEE Micro, vol 25, no. 3, pp. 110-117, May-Jun 2005.

[36] E. Gabrielyan, R.D. Hersch, "Network topology aware scheduling of collective communications", 10th International Conference on Telecommunications (ICT 2003), vol.2, pp. 1051-1058, Feb 2003.

[37] A, Gangwar, M. Balakrishnan, P. R. Panda, A. Kumar, "Evaluation of bus based interconnect mechanisms in clustered VLIW architectures", Proc. Design, Automation and Test in Europe (DATE 2005), vol. 2, pp. 730-735, Mar 2005.

[38] S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, K. De Bosschere, "System scenario based design of dynamic embedded systems", accepted for ACM Trans. on Design Automation for Embedded Systems (TODAES), vol. 14, 2009.

[39] M.I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, S. Amarasinghe, "A stream compiler for communication-exposed architectures", ACM SIGARCH Computer Architecture News, vol. 36, no. 5, pp. 291-303, Dec 2002.

[40] "GraphViz - Graph Visualization Software", `http://www.graphviz.org/`

[41] W.D. Grover, "Mesh-based survivable networks: options and strategies for optical, MPLS, SONET and ATM networking", Prentice-Hall, Upper Saddle River, 2004, chap. 4.

[42] R. Gruber, P. Volgers, A. De Vita, M. Stengel, "Commodity computing results from the Swiss-Tx project", Electronic Notes in Future Generation Computer Systems, Mar 2001.

[43] "DLX simulator in SystemC", http://systems.cs.colorado.edu/grunwald/Classes/CSCI5593

[44] "GSM 06.10 lossy speech compression", http://kbs.cs.tu-berlin.de/jutta/toast.html

[45] Design and analysis of arbitration protocols, "F. El Guibaly", IEEE Trans. on Computers, vol. 38, no 2, pp. 161-171, Feb 1989.

[46] J. Guo, A. Papanikolaou, P. Marchal, F. Catthoor, "Physical design implementation of segmented buses to reduce communication energy", Asia and South Pacific Conference on Design Automation (ASPDAC 2006), pp. 42-47, Jan 2006.

[47] J. Guo, A. Papanikolaou, P. Marchal, F. Catthoor, "Energy/area/delay trade-offs in the physical design of on-chip segmented bus architecture", International Workshop on System-Level Interconnect Prediction (SLIP 06), pp. 75-81, Mar 2006.

[48] J. Guo, A. Papanikolaou, F. Catthoor, "Topology exploration for energy efficient intra-tile communication", Design Automation Conference (ASPDAC 2007), pp. 178-183, Jan 2007.

[49] J. Guo, A. Papanikolaou, H. Zhang, F. Catthoor, "Energy/area/delay trade-offs in the physical design of on-chip segmented bus architecture", IEEE Trans. VLSI, vol. 15, no. 8, pp. 941-944, Aug 2007.

[50] J. Guo, "Analysis and optimization of intra-tile communication network", Doctorate thesis, Katholieke Universiteit Leuven, Aug 2008 .

[51] J. Guo, A. Papanikolaou, M. Stucchi, K. Croes, Z. Tokei, F. Catthoor, "A tool flow for predicting system level timing failures due to interconnect reliability degradation", Proc. of the 18th ACM Great Lakes symposium on VLSI, Orlando, FL, USA pp. 291-296 , 2008.

[52] J. Guo, A. Papanikolaou, M. Stucchi, K. Croes, Z. Tokei, F. Catthoor, "The analysis of system-level timing-failures due to interconnect reliability degradation", IEEE Trans. on Device and Materials Reliability, submitted.

[53] J. Hartmann, "Towards a new nanoelectronic cosmology", Solid-State Circuits Conference (ISSSC 2007), pp. 31-37, Feb 2007.

[54] J. Henkel, W. Wolf, S. Chakradhar, "On-chip networks: a scalable, communication-centric embedded system design paradigm", Proc. 17th International Conference on VLSI Design (VLSID 04), pp. 845-851, Aug 2004.

[55] J.L. Hennessy and D.A. Patterson, "Computer architecture: a quantitative approach, 2nd ed.", Morgan Kaufman, San Francisco, 1995.

[56] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Energy costs of transporting switch control bits for a segmented bus", Proc. 16th Annual Wsh. on Circuits, Systems and Signal Processing (ProRisc 2005), pp 359-364, Nov 2005.

[57] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, K. Debosschere, W. Philips, "Energy consumption for transport of control information on a segmented software-controlled communication architecture", 2nd Intl. Workshop on Applied Reconfigurable Computing (ARC 2006), LNCS 3985, p. 52-58, Mar 2006.

[58] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Network control, topology and transfer scheduling for synchronous system-on-chip communication", Architecture and Compilers for Embedded Systems (ACES) 2006, Ghent University, pp. 42-45, Oct 2006.

[59] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Using a linear sectioned bus and a communication processor to reduce energy costs in synchronous on-chip communication", Intl. Symp. on System-on-Chip (SOC 2007), p. 117-120, Jan 2008.

[60] K. Heyrman, "Using SystemC for the power simulator of an on-chip sectioned bus", `http://www-ti.informatik.uni-tue\-bingen.de/~systemc/Documents/Presentation-SM07-UP2_heyrman.pdf`, European System C Users Group Special Meeting (ESCUG-SM), Nov 2007, Tampere, Finland.

[61] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Control of low-power synchronous on-chip communication", Advanced Computer Architecture and Compilers for Embedded Systems (ACACES 2007), pp 29-32, Jul 2007.

[62] K. Heyrman, A. Papanikolaou, F. Catthoor, P. Veelaert, W. Philips, "Power gating for wires", IEEE Trans. on VLSI Systems, (accepted for publication).

[63] R. Ho, K.W. Mai, M.A. Horowitz, "The future of wires", Proc. IEEE, vol. 89, no. 4, pp. 490-504, Apr 2001.

[64] J. Hoogerbrugge, H. Corporaal, "Transport-triggering vs. operation-triggering", Springer, Berlin Heidelberg, 1994.

[65] M. Horowitz, T. Indermaur, R. Gonzalez, "Low-power digital design", IEEE Symp. on Low Power Electronics, 1994, pp. 8-11, Oct 1994.

[66] Cheng-Ta Hsieh, M. Pedram, "Architectural energy optimization by bus splitting", IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 21, no. 4, pp. 408-414, Apr 2002.

[67] "International Technology Roadmap for Semiconductors, 2007 Edition", `http://www.itrs.net/Links/2007ITRS/`

[68] M. Jayapala, "Low energy instruction memory organisation for embedded processors", Doctorate thesis, Katholieke Universiteit Leuven, Sep 2005 .

[69] A. A. Jerraya, W. Wolf ed., "Multiprocessor systems-on-chips", Morgan Kaufmann, San Francisco, 2005.

[70] W.-B. Jone, J.S. Wang, H.I. Lu, I.P. Hsu, J.-Y. Chen,, "Design theory and implementation of low-power segmented bus systems", ACM Trans. on Design Automation of Electronic Systems (TODAES), vol. 8 no. 1, pp. 38-54, Jan 2003.

[71] H. Kalte, D. Langen, E. Vonnahme, A. Brinkmann, U. Rückert , "Dynamically reconfigurable system-on-programmable-chip", Proc. 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (EUROMICRO-PDP'02), pp. 235-242, Jan 2002.

[72] J.T. Kao, A.P. Chandrakasan, "Threshold voltage techniques for low-power digital circuits", IEEE Journal of Solid-state Circuits, vol. 35 no. 7, pp. 1009-1018, Jul 2000.

[73] B.W. Kernighan, R. Pike, "The Practice of Programming", Addison-Wesley, Upper Saddle River NJ, 1999.

[74] P. Kuonen, "The K-ring", Proc. of the Eur. Research Seminar in Advances in Distributed Systems (ERSADS), Apr 1995.

[75] K. Lahiri, A. Raghunathan, S. Dey, "Design space exploration for optimizing on-chip communication architectures", IEEE Trans. CAD of ICS, vol. 23, no. 6, pp. 952-961, Jun 2004.

[76] V. Lahtinen, E. Salminen, K. Kuusilinna, T. Hamalainen, "Comparison of synthesized bus and crossbar interconnection architectures", Proc. of the 2003 International Symposium on Circuits and Systems (ISCAS '03), vol. 5, pp. 433-436, May 2003.

[77] Y. Li and S.Q. Zheng, "Prefix computation using a segmented bus", 28th Southeastern Symposium on System Theory, pp. 416-420, Apr 1996.

[78] J. Liu et al., "Memory system compression and its benefits", J. Liu, N.R. Mahapatra, K. Sundaresan, S. Dangeti, B.V. Venkatrao, 15th Annual IEEE International ASIC/SOC Conf, p. 41-45.

[79] V. Lyuboslavsky, "Segmented bus", CSE 530 course, Pennsylvania State Unversity, 2000.

[80] L. Macchiarulo, E. Macii, M. Poncino, "Low-energy encoding for deep-submicron address buses (ISLPED 2001), pp. 176-181", Aug 2002,

[81] R. Mäkelä, J. Takala, O. Vainio, "Analysis of different bus structures for transport triggered architecture", Proc. IEEE NORCHIP Seminar 2003, pp. 56-59, Nov 2003.

[82] G. De Micheli, "Synthesis and Optimization of Digital Circuits", Mc Graw-Hill, New York, 1994.

[83] L. M. Ni, "Issues in designing truly scalable interconnection networks", Proc. of the 1996 Workshop on Challenges for Parallel Processing , pp. 74-83, Aug 1996.

[84] J. M. Nielsen, "On the number of maximal independent sets in a graph", Alcom-FT Technical Report Series ALCOMFT-TR-02-87, 2002.

[85] T. Noll, "Low-power arithmetics for SoC", Intl. Symp. on System-on-Chip (SOC 2007), Nov 2007, Tampere, Finland.

[86] J. Nurmi ed., "Processor design: system-on-chip computing for ASICs and FPGAs", Springer, Dordrecht, 2007.

[87] M. Ohlidal et al., "Performance of collective communications on interconnection networks with fat nodes and edges", Proc. of the Intl. Conf. on Networking, Intl. Conf. on Systems and Intl. Conf. on Mobile Communications and Learning Technologies (ICN/ICONS/MCL 006), pp. 32-32, Apr 2006.

[88] D. A. Grable, A. Panconesi, "Nearly optimal distributed edge colouring in O(log log n) rounds", Proc. of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 97), 1997.

[89] A. Papanikolaou, K. Koppenberger, M. Miranda, F. Catthoor, "Architectural and physical design optimizations for efficient intra-tile communication", Proc. IEEE Workshop on Signal Processing Systems, pp. 176-181, Oct 2004 .

[90] A. Papanikolaou, "Segmented buses (design-time communications network): what, why and how?", IMEC internal document, Dec 2004 .

[91] A. Papanikolaou, F. Starzer, M. Miranda, F. Catthoor, K. De Bosschere, "Architectural and physical design optimizations for efficient intra-tile communication", Proc. International System-on-Chip Symposium (SoC 2007), pp. 112-115, Nov 2005..

[92] A. Papanikolaou, "Application-Driven Software Configuration of Communication Networks and Memory Organizations", Doctorate thesis, Ghent University, Dec 2006 .

[93] A. Papanikolaou, H. Wang, M. Miranda, F. Catthoor, W. Dehaene, "Reliability issues in deep deep sub-micron technologies: time-dependent variability and its impact on embedded system design", book chapter in "VLSI-SoC: research trends in VLSI and Systems-on-Chip", Springer, pp. 119-141, 2008.

[94] S. Pasricha, N. Dutt, "On-chip communication architectures – System on chip interconnect", Morgan Kaufman, San Francisco, 2008.

[95] M. Pedram and J. Rabaey ed., "Power aware design methodologies", Kluwer, Dordrecht, 2002, chap. 13.

[96] "Property Syntax Language Reference Manual", www.eda.org/vfv/docs/PSL-v1.1.pdf

[97] J. Rabaey, "Digital integrated circuits: a design perspective (2nd edition)", Prentice Hall, Englewood Cliffs NJ, 2003.

[98] V. Raghunathan, M.B. Srivastava, R.K. Gupta, "A survey of techniques for energy efficient on-chip communication", Design Automation Conference (DAC 2007), pp. 900-905, Jun 2003.

[99] P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, D. Verkest, "Distributed loop controller architecture for multi-threading in unithreaded VLIW Processors", Proc. 9th ACM/IEEE Design and Test in Europe Conf. (DATE'06) pp. 339-344, Mar 2006.

[100] P. Raghavan, A. Lambrechts, J. Absar, M. Jayapala, F. Catthoor, "Playing the trade-off game: Architecture exploration using COFFEE", ACM Trans. on Design Automation for Embedded Systems (TODAES), to be published.

[101] P. Raghavan, A. Lambrechts, J. Absar, M. Jayapala, F. Catthoor, D. Verkest, "COFFEE: Compiler framework For energy-aware exploration", Proc. Intnl. Conf. on High-Perf. Emb. Arch. and Compilers (HIPEAC'08), pp. 193-208, Jan 2008.

[102] A Comparison of Five Different Multiprocessor SoC BusA , "Euromicro Symposium on Digital Systems Design (DSD'01), pp. 0202", K. K. Ryu, E. Shin, V. J. Mooney, 2001.

[103] E. Salminen, V. Lahtinen, K. Kuusilinna, T. Hamalainen, "Overview of bus-based system-on-chip interconnections", Proc. of the 2002 International Symposium on Circuits and Systems (ISCAS '02), vol. 2, pp. 372-375, May 2002.

[104] C. Sanz Pineda, M. Prieto, I. Gomez, A. Papanikolaou, M. Miranda, F. Catthoor, "Combining system scenarios and configurable memories to tolerate unpredictability", ACM Trans. on Design Automation for Embedded Systems (TODAES), vol. 13, no. 3, art. 49, pp. 1-7, Jul 2008.

[105] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli, "Addressing the system-on-a-Chip interconnect woes through communication-based design ", 38th Conference on Design Automation (DAC 01), pp. 667-672, Jun 2001.

[106] J.G. Siek, L-Q Lee, A. Lumsdaine, "The Boost graph library: User Guide and Reference Manual", Addison-Wesley, Upper Saddle River NJ, 2002.

[107] A. Stepanov, "Foreword to the STL Tutorial and Reference Guide, 2nd Ed., by D.R. Musser, G.J. Derge and A Saini", Addison-Wesley, Upper Saddle River NJ, 2001.

[108] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, L. Benini, "pipes Lite: a synthesis oriented design library for networks on chips", Conference on Design, Automation and Test in Europe (DATE 05), vol. 2, pp. 1188-1193 , Mar 2005.

[109] J. Suh, E-G. Kim, S. P. Crago, L. Srinivavsan, M. C. French, "A performance analysis of PIM, stream processing and tiled processing on memory-intensive signal processing kernels", ACM SIGARCH Computer Architecture News, vol. 31, no. 2, May 2003.

[110] "Trimaran, An Infrastructure for Research in Backend Compilation and Architecture Exploration", `www.trimaran.org`

[111] T. Vander Aa, "Instruction transfer and storage exploration for low energy embedded VLIWs", Doctorate thesis, Katholieke Universiteit Leuven, Sep 2005 .

[112] S. Virtanen, T. Nurmi, J. Paakkulainen, J. Lilius , "A system-level framework for designing and evaluating protocol processor architectures", International Journal of Embedded Systems, vol. 1, no. 1-2, pp. 78-90, 2005.

[113] T. van Meeuwen, A. Vandecappelle, A. van Zelst, F. Catthoor, D. Verkest, "System-level interconnect architecture exploration for custom memory organisations ", 14th International Symposium on System Synthesis (ISSS 2001) Proc., pp. 13-18, 2001.

[114] Vitaly Voloshin, "Coloring Mixed Hypergraphs: Theory, Algorithms and Applications", AMS, Providence, 2002, chap.12.

[115] "W. S. Voon, Combinations in C++, Part 2", `http://www.codeproject.com/KB/recipes/CombC.aspx`

[116] H. Wang, A. Papanikolaou, M. Miranda, F. Catthoor, "A global bus power optimization methodology for physical design of memory dominated systems by coupling bus segmentation and activ-

ity driven block placement", Design Automation Conference (ASP-DAC 2004), pp. 759-761, Jan 2004.

[117] H. Wang, "Word and decoder organization methodology for the generation of energy and delay trade-offs in embedded SRAMS", Doctorate thesis, Katholieke Universiteit Leuven, Jun 2007.

[118] D.B. West, "Introduction to Graph Theory", Prentice-Hall, Upper Saddle River, 2001.

[119] M.V. Wilkes, J.B. Stringer, "Microprogramming and the design of the control circuits in an electronic digital Computer", Proc. of the Cambridge Philosoph. Soc., 1953.

[120] "WISHBONE SoC Interconnection Architecture for Portable IP Cores - Rev B.3, Sep 7, 2002", www.opencores.org

[121] B.P. Wong, A. Mittai, Y. Cao, G. Starr, "Nano-CMOS circuit and physical design, chap. 8", Wiley, Hoboken NJ, 2005.

[122] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, R. Lauwereins, "Managing dynamic concurrent tasks in embedded real-time multimedia systems", Proc, of the 15th International Symp. on System Synthesis (ISSS'02), p. 112-119 , Oct, 2002 .

[123] Y. Zhang, M. J. Irwin, "Power and performance comparison of crossbars and buses as on-chip interconnect structures", Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers, 1999, vol. 1, pp. 378-383, Oct 1999.

[124] H. Zhang, M. Wan, V. George, J. Rabaey, "Interconnect architecture exploration for low-energy reconfigurable single-chip DSPs", IEEE Computer Society Workshop on VLSI (VLSI 99), pp. 2-8, Apr 1999.

[125] L. Zhong, N. K. Jha, "Interconnect-aware high-level synthesis for low power ", Computer Aided Design, (ICCAD 2002), pp. 110-117, Nov 2002 .

Concise Index