ON RUN-TIME EXPLOITATION OF CONCURRENCY

PHILIP K.F. HÖLZENSPIES

On run-time exploitation of concurrency

Philip K.F. Hölzenspies

Members of the dissertation committee:

Prof. dr. ir.	G.J.M. Smit	University of Twente (promoter)
Prof. dr.	J.L. Hurink	University of Twente (promotor)
Dr. ir.	J. Kuper	University of Twente (assistant-promotor)
Prof. dr. ir.	Th. Krol	University of Twente
Prof. dr.	J.C. van de Pol	University of Twente
Prof. dr.	A. Shafarenko	University of Hertfordshire
Prof. dr.	Ch. Jesshope	University of Amsterdam
Prof. dr. ir.	A.J. Mouthaan	University of Twente (chairman and secretary)

Cover design by Diederik Telman. This thesis was printed by Gildeprint, The Netherlands.

ISBN 978-90-365-3021-7
DOI 10.3990/1.9789036530217

On run-time exploitation of concurrency

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 23 april 2010 om 13.15 uur

door

Philip Kaj Ferdinand Hölzenspies

geboren op 29 april 1980
te Houten

Dit proefschrift is goedgekeurd door:

prof. dr. ir.	G.J.M. Smit	(promotor)
prof. dr.	J.L. Hurink	(promotor)
dr. ir.	J. Kuper	(assistent-promotor)

Voor mijn ouders,

Trix en Bert Hölzenspies

Abstract

The 'free' speed-up stemming from ever increasing processor speed is over. Performance increase in computer systems can now only be achieved through parallelism. One of the biggest challenges in computer science is how to map applications onto parallel computers.

Concurrency, seen as the set of valid traces through a program, is utilized by translating it into actual parallelism, i.e. into the simultaneous execution of multiple computations. With higher degrees of unpredictability—both with regards to the actual workload and to the availability of resources—more can be gained from making scheduling and resource management decisions at run-time, when more information (such as resource availability and required Quality of Service (QoS) level) is available. In cases where concurrency is data-dependent, programming models and their supporting run-time systems also benefit from exposing concurrency when that data is known, viz. at run-time. In this thesis, two systems for run-time exploitation of concurrency are discussed.

The first system discussed in this thesis is an on-line spatial resource manager for real-time streaming applications, especially in energy constrained environments. In embedded systems, these applications typically require QoS guarantees, are structurally stable (do not change over time) and are active for a (relatively) long period of time. With increasing complexity, embedded systems consist increasingly of many independent processors with varying degrees of specialization. Designing systems in such a way is beneficial for flexibility, yield increase and energy conservation. However, exploiting such a heterogeneous multi-processor system in order to realize these benefits requires that the resources it provides are dynamically assigned to applications.

A formal and precise definition of this on-line spatial resource management problem is given in this thesis and qualitative evaluation criteria by which on-line spatial resource managers can be compared are introduced. Constraints on applications and techniques for their modelling are discussed. Since the complexity of this problem is prohibitive and the time constraints to make choices are tight, a heuristic approach is introduced. In this approach, the complete problem of spatial resource management is partitioned into the subproblems of binding, mapping, routing, and QoS validation.

The subproblems are ordered in the sense that choices made for the solutions to

earlier subproblems are considered fixed when solving later subproblems. Since the subproblems still have a high complexity, algorithms and approaches from literature are adapted to partition them further. The adapted algorithms are implemented in Kairos, a proof-of-concept on-line spatial resource manager for heterogeneous multi-processor systems. A large use case, taken from a state-of-the-art industrial application, is used to explore Kairos' capabilities. With this use case and a with synthetic benchmark, Kairos is shown to be a successful proof-of-concept implementation for on-line spatial resource management and, thus, the problem is shown to be solvable with acceptable concessions.

The second system discussed in this thesis deals with applications for which it is hard or even impossible to predict their behaviour to the extent that is necessary to fulfil real-time requirements. In particular, this holds for applications for which the amount of concurrency is highly data-dependent and the work done by different tasks in an application is unbalanced, variable and unpredictable. For these applications, performance can not be guaranteed, but by exposing (data-dependent) concurrency at run-time, an application's performance and the total system's utilization can be improved.

The system discussed here is SNet. It is developed at the University of Hertfordshire and comprises a coordination language, a programming model and a run-time system. A great strength of SNet is that it allows for the separation of concerns between application engineering and concurrency engineering. The application engineer does not program individual threads with their synchronization and communication, but decomposes the application into small units of work on a stream of input data. In this thesis, a denotational semantics for SNet is presented with proof that under those semantics, SNet is prefix monotonic, i.e. for every finite prefix of the input stream, a prefix of the output stream exists that is unchanged by further input. Furthermore, a novel execution model is presented that exposes significantly more concurrency than the former execution model. A strong indication is given that a schedule exists, such that the novel execution model does not introduce non-termination.

Samenvatting

Dat het versnellen van centrale rekeneenheden onze toepassingen blijft versnellen mag niet langer voor lief worden genomen. Betere prestaties van nieuwe computers zullen noodgedwongen voort moeten komen uit parallellisme. Eén van de grootste uitdagingen van de hedendaagse informatica is hoe onze toepassingen af te beelden op parallelle computers.

Multiprogrammering (uitgedrukt in het aantal geldige ordeningen van instructies) kan worden omgezet in parallellisme of, met andere woorden, in gelijktijdige uitvoeringen van meerdere instructies. Nu in nieuwe computersystemen en toepassingen zowel de werklast als de beschikbare middelen onderhevig zijn aan toenemende onvoorspelbaarheid, valt er veel te winnen met het uitstellen van beslissingen over toewijzingen van middelen aan toepassingen totdat er meer bekend is over beiden. Dit is bijvoorbeeld het geval in lopende systemen. Zeker wanneer de mate van multiprogrammering afhangt van waarden, kunnen systemen beter worden benut door beslissingen over toewijzingen te nemen wanneer de relevante waarden bekend zijn. Wederom is dit het geval in lopende systemen. In dit proefschrift worden twee systemen beschreven die multiprogrammering benutten op het moment of nadat toepassingen worden gestart.

Het eerstbeschreven systeem is dat van het beheer van ruimtelijke middelen in lopende computersystemen voor stroomverwerkende toepassingen met harde tijdseisen, met bijzondere aandacht voor energiebegrensde omgevingen. In ingebedde systemen geldt voor zulke toepassingen doorgaans dat ze een onveranderlijke ruimtelijke structuur hebben, dat ze harde garanties nodig hebben aangaande de kwaliteit van het resultaat (in termen van tijdigheid) en dat ze veelal voor lange tijd lopen. Met de toenemende complexiteit bestaan ingebedde systemen in toenemende mate uit een groot aantal onafhankelijke rekeneenheden (en andere middelen, zoals geheugens) met wisselende specialisatiegraad. Deze ontwerptrend kan bijdragen aan de flexibiliteit van computersystemen, aan de opbrengst van de productie van geïntegreerde schakelingen en aan energiebesparing. Beheer van ruimtelijke middelen onder draaitijd is echter noodzakelijk om systemen, bestaande uit een grote heterogene verzameling rekeneenheden (en andere middelen), toepassingen uit te laten voeren met de vereiste garanties.

Wat wel en niet omvat wordt door beheer van ruimtelijke middelen onder draaitijd wordt formeel gedefinieerd in dit proefschrift. Daarnaast worden enkele kwalita-

tieve criteria om beheermethoden te kunnen vergelijken gegeven en zijn eisen die aan toepassingen gesteld worden en technieken om zulke toepassingen te modelleren omschreven. Om in de zeer korte hiervoor beschikbare tijd toewijzingskeuzes te maken en met het oog op de complexiteit van dit probleem worden heuristieken ingevoerd. Hierbij wordt het totale probleem van beheer van ruimtelijke middelen opgedeeld in vier deelproblemen, te weten binding, afbeelding, routering en tijdseisenvalidatie.

Deze deelproblemen moeten als geordend worden beschouwd. Dit betekent dat oplossingen voor eerdere deelproblemen als gegeven worden beschouwd bij de oplossing van latere deelproblemen. Daar de individuele deelproblemen zelf nog steeds zeer complex zijn worden algoritmen uit de literatuur verder gespecialiseerd om ze verder op te delen en op te lossen. Het geheel van al deze algoritmen is bij wijze van demonstratie van de haalbaarheid geïmplementeerd in Kairos. Kairos wordt getoetst met behulp van een grote industriële voorbeeldtoepassing en met gesynthetiseerde toepassingen. De resultaten tonen aan dat met Kairos voldoende is gedemonstreerd dat het probleem oplosbaar is met aanvaardbare concessies.

Het voortsbeschreven systeem faciliteert toepassingen, waarvoor het moeilijk of zelfs onmogelijk is om het gedrag (met betrekking tot behoeften aan middelen en tijd) te voorspellen. Deze moeilijke voorspelbaarheid komt vooral veel voor bij toepassingen, waarvan de mate van multiprogrammering afhankelijk is van de waarden van de invoer en waar de mogelijke opdeling van werk in deeltaken onregelmatig, wisselend en onvoorspelbaar is. De prestaties van dergelijke toepassingen kunnen niet worden gegarandeerd, maar zijn wel te verbeteren door (waarde-afhankelijke) multiprogrammering op draaitijd te identificeren en uit te buiten, waardoor ook de efficiëntie van het computersysteem dat de toepassing uitvoert wordt verbeterd.

Het betreffende systeem is het van de University of Hertfordshire afkomstige SNet. Het omvat een coördinatietaal, een programmeermodel en een draaitijdomgeving. De kracht van SNet schuilt in de scheiding van het ontwerp van de toepassing enerzijds en de multiprogrammering anderzijds. De ontwikkelaar van de toepassing hoeft zich niet met fijnmazige synchronisatie tussen deeltaken bezig te houden, maar slechts de gehele toepassing op te delen in kleine eenheden van werk, gedefiniëerd op een gegevensstroom. In dit proefschrift wordt een denotationele semantiek gegeven van de taal SNet en met die semantiek wordt een bewijs gegeven dat de taal SNet prefixmonotoon is in de invoer, d.i. dat voor iedere eindige prefix van de invoer er een prefix van de uitvoer bestaat die niet afhangt van verdere invoer. Ten slotte wordt er een nieuw executiemodel beschreven, dat een significant hogere graad van multiprogrammering blootlegt dan het huidige executiemodel van SNet. Voor dit executiemodel wordt een sterke indicatie gegeven, dat er altijd een schema kan worden gevonden waardoor er geen nonterminatie wordt geïntroduceerd.

Dankwoord

Gezien de context van dit dankwoord, is het gepast te beginnen bij mijn promotoren Gerard Smit en Johann Hurink en assistent-promotor Jan Kuper.

Door Gerard ben ik terechtgekomen bij de groep die tijdens mijn onderzoek de zijne werd. Hij stelt iedereen in de groep in staat naar eigen inzicht te werken, maar weet daarbij wel een hechte groep te behouden, waarin discussie en kruisbestuiving centraal staan. Nergens anders heb ik groepen gezien zo groot als CAES, die toch zo betrokken zijn. Toen ik begon met mijn onderzoek, werd Gerards vooruitziende blik door velen in de vakgemeenschap aan het begin van mijn onderzoek nog met scepsis bekeken, maar in ieder geval voor wat betreft mijn onderzoek blijkt hij—tot mijn grote genoegen—gelijk te krijgen.

Johann heb ik pas leren kennen tijdens mijn aanstelling als AiO. Van zijn kennis en doortastende inzicht heb ik van het begin af aan erg genoten, maar de misschien nog wel belangrijkere onderlinge sfeer is goed begonnen en de afgelopen jaren alleen maar vooruit gegaan. Vooral tegen het einde ben ik zeer onder de indruk geraakt van hoe makkelijk hij voornoemd inzicht aanwendt voor sociale en politieke doeleinden.

Jan weet ongetwijfeld zelf wel dat ik hier niet alles kan benoemen. Ruimschoots voor mijn AiO-tijd rekende ik hem al tot mijn goede vrienden en dat is in zijn rol als begeleider meer dan eens bevestigd. Of onze discussies nu over wiskunde, programmeren, onderwijs, filosofie, whisky, muziek of mensen gaan, ik put er altijd energie en inspiratie uit. Dat we het niet eens zullen worden over schrijfstijl (Over hoeveel losse zinnen had ik voorgaande opsomming uit moeten spreiden?) en voetbal, daar heb ik vrede mee. Ik hoop nog vele jaren plezier van ons contact te hebben.

Evenzo onmisbaar bij een promotie zijn de paranimfen. Het is bijna niet meer voorstelbaar dat ik Vincent Jeronimus leerde kennen als mijn leidinggevende bij een bijbaan naast mijn studie. Al snel bleek dat wij het buiten het werk goed met elkaar konden vinden, vooral toen ik bij hem in een band belandde die het nog negen jaar vol zou houden. Momenteel is het even gedaan met onze bands, maar samen muziek maken is nog zeker niet voorbij. Timon ter Braak ken ik nog niet zo lang, maar als afstudeerder heeft hij mij meerdere malen verrast met goede inzichten. Naast het feit dat hij van aanpakken weet en snel zijn eigen en mijn ideeën realiseert, is hij een zeer aangename reisgenoot gebleken bij workshops en

conferenties. De overgang van afstudeerder naar collega was een zeer natuurlijke, evenals zijn betrokkenheid bij mijn promotie als paranimf. Timon, jouw bijdragen aan mijn onderzoek hebben mijn zo nu en dan gestrande motivatie al meerdere keren vlotgetrokken.

Natuurlijk is de gehele CAES-groep van grote invloed geweest op mij gedurende de jaren die aan dit proefschrift vooraf gingen. Ik wil enkele mensen echter specifiek bedanken voor hun invloed en contact.

Enkele jaren heb ik het genoegen gehad kantoorgenoot te zijn geweest van Pascal Wolkotte. Pascal, met veel plezier heb ik je 'besmet' met een passie voor typografie en allerlei daaraan gerelateerde software. Met nog veel meer plezier heb ik over diezelfde typografie en software, maar zeker ook over data-analyse en onderzoekshouding, veel van je geleerd. Discussies over alles in en ver buiten de techniek mag ik nog altijd graag met je voeren.

Ten tijde van dit schrijven deel ik het kantoor met Mark Westmijze; eerder al als huisen nu als kantoorgenoot heb ik veel plezier met hem gehad. De avondvullende (en te zeldzame) gesprekken met Albert Molderink zijn altijd boeiend en het wekelijkse squashen is iedere keer een moment om te aarden en even goed adem te halen. Tot slot is een leerstoel reddeloos verloren zonder goede secretaresses. Dit ben ik nóg meer gaan waarderen toen ik een paar maanden te gast was bij een groep waar zulke ondersteuning ontbrak. De CAES groep mag zich in het bijzonder gelukkig prijzen met de secretaresses die ze heeft, te weten Marlous Weghorst, Nicole Baveld en Thelma Nordholt.

Aforementioned visit to another group was Alex Shafarenko's group at the University of Hertfordshire. I thank all the people I met there for a wonderful visit. Not only did I have a great time, but I learned quite a few things and left with thoroughly replenished inspiration.

Natuurlijk is er de afgelopen jaren een leven, zij het beperkt, naast het promotieonderzoek geweest. Jan Koornstra is al een klein decennium mijn muzikale baken en geweten. Jans geduld, zeer brede muzikale interesse en uitzonderlijke gevoel voor mijn vaak ongecontroleerde bijdragen zijn zeldzaam te noemen. De diepgang van onze woordeloze communicatie heb ik bij niemand anders nog ervaren. Naast al onze wisselende muzikale bezigheden spelen wij jaarlijks een huiskamerconcert bij Anja & Jan Wagner. Deze jaarlijkse gelegenheid en de daarbij betrokken mensen zijn voor mij van grote betekenis.

In communicatie met anderen staat het woordelijke juist vaak centraal. Discussies met Robert Nijssen, Pièrre Jansen en Rien Boone over politieke, maatschappelijke en levensbeschouwelijke vraagstukken zijn zeer prikkelend. Daarbij heeft het gezin van laatstgenoemde mij altijd zeer hartelijk ontvangen en op verschillende momenten en manieren ondersteund. Met Martin Bosker spreek ik ook graag over een verscheidenheid aan thema's, maar waar ik Martin vooral dankbaar voor ben, is dat hij mij heeft laten zien dat het maken van foto's niet vervelend hoeft te zijn en vaak zelfs leuk is. Hij nam de foto die verwerkt is in de kaft van dit proefschrift en

de uitnodiging voor de promotieplechtigheid, maar er gingen er velen aan vooraf. Martin kan ik slechts zeggen: 9–0! Dennis Mulder is ook vaak te vinden voor uitgebreide gesprekken ver na zonsondergang. Daarbij heeft hij zijn expertise op taalgebied aan willen wenden om delen van dit proefschrift te becommentariëren.

Diederik Telman ontwierp kaft van en uitnodiging bij dit proefschrift. Hij staat ver van mijn vakgebied, maar heeft na een korte uitleg de inhoud blijkbaar zo goed bevat, dat hij met een geweldige abstracte representatie op de proppen kwam. Naast zijn grote grafische talent heb ik ook erg van zijn ritmische talent genoten, omdat hij jaren bandgenoot is geweest. Andere bandgenoten die ik zeer dankbaar ben voor de muzikale samenwerking zijn o.a. Stefan Klein, Daniël van Doorn, Ivo Kreetz, Ties Brands, Joris Holtackers en alle sessiemuzikanten van De Cactus.

Ik prijs mij gelukkig met de velen waarbij ik in tijden van zwaar weer kan schuilen, maar met wie het in tijden van voorspoed evenzo goed toeven is. Het zijn er teveel om een uitputtende opsomming te geven en dus moet ik mij beperken tot de zeer uitzonderlijke gevallen. Léon & Angela Buijs, Paula den Boer & Alex Kok, Maarten van der Weg, Erik Hagreis, René Beerens, Pascal & Marieke Viskil, Addy Viskil, Duco Hoogland, Rik Bos, Stefan Janssen, Bertus Klein, Nelleke Ruijter, Pascal Huis in 't Veld en Ivo Belt vallen allemaal in deze categorie.

Als laatste wil ik hen bedanken die ik al het langst in mijn leven heb: Met zus Laura deel ik de laatste jaren een steeds verder overlappende muziekinteresse. Laura, onze bezoekjes doen mij altijd veel goed. Mijn broer Jurriaan is vaker dan hij weet een voorbeeld voor me geweest en zijn invloed door discussie en commentaar komt in dit proefschrift veel terug. Ten slotte wil ik mijn ouders bijzonder bedanken. Niet alleen hebben zij mij altijd ondersteund in materiële, maar vooral ook in immateriële zin. Ze staan altijd klaar met advies en wanneer het advies op is, is er altijd een onvoorwaardelijk thuis. Hoe bijzonder en belangrijk het voor me is dat ze allebei getuigen kunnen zijn van mijn promotie kan ik niet in woorden vatten.

Philip Hölzenspies
Hengelo, april 2010

Contents

Introduction

1.1 A truly new era for programmers

Computer science is a very young science. The discipline of programming is even younger. However, with the new millennium, there is one challenge that no programmer can safely ignore: how to program parallel computers. We can speak of a new era, not because the challenge itself is new, but because most programmers did not need to face it, until now. In the days of the first electronic computers, the programmer and the user were the same person, usually even at the same time. Computers were commonly developed with a specific application in mind. Two very important things have happened since, that changed this perspective on the design and use of computers: High-level programming languages have been developed and the market for consumer products has come to dictate the direction of computing. Both topics are addressed briefly here as reasons why mainstream programmers have been lead away from having to think about the internals of the computer running their programs and, thus, from having to deal with properties of parallel computers.

1.1.1 High-level programming languages

In the 1950s, the idea of machine-independent or high-level programming languages emerged. The first generally recognised complete compiler of a high-level programming language was IBM's FORTRAN compiler, developed by a team lead by John Backus, published in 1957 [7]. Shortly thereafter (1960) the first program,

written in COBOL, was compiled for two different computers [60]. Since then, programmers have arguably seen a consistent relaxation of constraints stemming from computer architectures. That is to say, the complexity of programming has been driven by a desire to tackle more complex problems and to improve extra-functional properties of source code (e.g. modularity), rather than by problems resulting from the evolution of underlying hardware.

There have been major hurdles on the ongoing road towards higher performance. However, these hurdles have been overcome by landmark achievements *under* the programming level: Integrated Circuit (IC) production technology, microprocessor architecture and compiler technology. As ICs have grown (in terms of numbers of transistors) and sped up (in terms of clock frequency) the power consumption has increased, giving rise to power dissipation (i.e. heat) problems. Also, with the increase of digital functionality in mobile devices (since the early 1990s), battery life became a concern. Energy problems have been solved in the past by, for example, moving from N-type Metal-Oxide-Semiconductor (NMOS) designs to designs based on Complementary Metal-Oxide-Semiconductor (CMOS), using copper instead of aluminium and Silicon-On-Insulator (SOI) technology. To reduce the time hardware components were idling and to increase the overall instruction throughput, architectural concepts have been developed that are still pervasive in modern computer architecture, like instruction pipelining, spooling, Direct Memory Access (DMA), superscalar, out-of-order & speculative execution and Single Instruction Multiple Data (SIMD). The performance gap between processors and memory has been narrowed by (among others) caches and pipelining memory architectures—e.g. Fast Page Mode (FPM) and Double Data Rate (DDR). The added programming complexities introduced by these architectural solutions (esp. resource contention in superscalar and cache-miss penalties) have been solved by compiler improvements.

1.1.2 CONSUMER AT THE HELM

In the previous section we described why programmers did not have to think about parallelism. However, there is also a good reason why programmers are now forced to: The market demands it.

As stated before: Early electronic computers were specifically designed for a particular application. Although quite a few notable computers were developed for a more general purpose (e.g. IBM S/360 & PDP-11), the most important milestone on the road to truly application agnostic computers was the introduction of the microprocessor in the 1970s. During the first three decades of the microprocessor era, there was an explosive growth of the number of architectures and instruction sets [100]. Although the x86 processor family was already very popular in the 1990s, its most important market share still was low-end office applications and consumers. Most applications considered 'high-end' or 'industrial' were run on other architectures, e.g. MIPS, POWER and SPARC. This has changed in the first decade of the twenty-first century. The x86 processor family has conquered the high-end market. Of the

top 500 supercomputers in the world, as of November 2009, 87.6% are built with x86 processors and 10.4% are based on POWER processors [94]. Workstations for industrial applications have seen a similar trend.

These market observations are important, because they are indicative of the fact that the computer landscape is determined predominantly by the *consumer* market. This has important implications for the world of applied computer science as well. Most significantly for the work in this thesis: It means that the consumer market is the initiator of the new era for programmers; one, in which programmers need to think about computers as parallel machines. There has been keen interest in parallel computing from academia and other researchers since, at least, the late 1950s. Many parallel computers have been built in the second half of the twentieth century, but until the turn of the century, parallel computing was considered by most to be something for the High Performance Computing (HPC) market only. At the start of the 2010s, it is hard to find off-the-shelf x86 processors that contain only one core. The x86 processor manufacturers have decided the time has come to move a responsibility for further speed-up to the programmer.

1.1.3 THE NEW ERA

There is a reasonable consensus that mainstream programming must move with the times. The prominent C++ expert Herb Sutter provocatively stated that "the free lunch is over" [93]. Asanovic et al. at University of California Berkley published an often cited vision statement identifying seven key questions for future parallel computing research, two of these deal with programming challenges [6]. Considerably less consensus exists about what the best approach is for programming parallel computers.

Concurrency vs Parallelism

One common source of disagreement among researchers is the precise definition of terminology. For a good understanding of the title of this thesis, one such terminology problem is especially relevant: concurrency and parallelism. Many (acceptable) definitions exist for these terms. However, in this thesis, these terms are used as follows:

- *Concurrency* is non-determinism with regards to the order in which events may occur.
- *Parallelism* is the degree to which events occur simultaneously.

Informally, concurrency can thus be seen as *potential* parallelism, i.e. if the order of events a and b is undefined, they may occur as a followed by b, as b followed by a, or simultaneously. Only in the last case, there is *actual* parallelism. In this terminology, a key difference between concurrent programming and parallel programming is

that the latter specifies a precise schedule of events. Parallel programming is only possible when resource availability is known precisely and the potential parallelism is not dependent on the program's input. If resource availability is known at compile-time, concurrent programs can be rewritten to parallel programs by a compiler. The central theme of this thesis is how postponing the translation from concurrency into parallelism until run-time can be beneficial, either for energy conservation or for performance optimization.

1.2 Approach and contributions of the thesis

There are programming disciplines in which concurrency has long ago been recognized as a necessity. Two of these disciplines are embedded systems and HPC. In this thesis, we examine these two disciplines and try to extend them towards mainstream programming.

In embedded systems, hardware is typically developed or configured for a specific application or a class of applications. This allows for very fine grained optimization in the design process. However, many embedded devices are pushed towards being multi-purpose platforms. Two examples of this are smart phones and automotive integrated multi-media systems. In traditional design approaches for embedded systems, such systems are programmed including detailed resource management/scheduling. Since both the number of applications and the complexity of the hardware are rapidly increasing, the complexity of such approaches becomes prohibitive. Thus, performing resource management in a running system reduces time-to-market and increases the overal flexibility. The first contribution of this thesis is a system for *on-line spatial resource management* that gives application developers a perspective of a (more) general purpose platform, instead of a complex system of individual shared resources. This work is introduced in more detail in section 1.3.

The central problem in programming for HPC is how to translate a large computational problem into a program that balances computational load over the available resources. Many applications have strongly data-dependent resource requirements and concurrency. Parallel computers are becoming more unpredictable as well, since an increasing number of HPC systems are based on clusters and clouds [94]. Application engineers (physicists, chemical scientists, etc.) understand the complexity and the structure of the problem very well. Concurrency engineers have that kind of understanding of distributed computing with a complex structure of interconnected resources. The problem is, that both the application and the parallel machine need to be understood well, to produce a program that delivers the desired high performance. In an attempt to separate these concerns, researchers at the University of Hertfordshire have developed a *coordination language* called SNet [37]. In SNet, the structure of application is specified in such a way that a run-time system can make resource management choices to try to deliver the highest performance with the available resources. The second contribution of this thesis consists of two

contributions to SNet: A denotational semantics for the language SNet is given, as well as a new execution model, implemented in a compiler and run-time system. This work is introduced in more detail in section 1.4.

1.3 On-line spatial resource management

Very few embedded systems are designed from scratch. Instead, they are generally constructed by combining Intellectual Property (IP) blocks. To create a system with IP blocks, they must be able to communicate. This is done with some form of interconnection. Because embedded systems are typically cost and energy constrained, IP blocks are usually constructed around one central bus. This means that interconnection introduces a notion of spatial locality to resource management. Taking this locality into account can help increase QoS and energy efficiency, as discussed below.

Terminology in the area of on-line spatial resource management is not (yet) very stable. Originally, we referred to this research as run-time spatial mapping, which is the prevalent terminology in most of the publications related to this thesis. This has caused debate and confusion in the past. *Run-time* is often associated with the run-time of an application, rather than that of a system. Similarly, *mapping* is a term used in many different ways in different areas. *Resource management* is the generic term for allowing or refusing applications access to resources. As explained above, these resources are arranged in a way that their *spatial* properties are relevant to resource management. The qualifier *on-line* indicates, that spatial resource management occurs in a running system.

1.3.1 real-time streaming applications

Real-time streaming applications are implemented and used in portable and otherwise energy constrained (embedded) systems. Such systems require energy-aware tools and an energy-efficient processing architecture. Typical examples of such applications involve Digital Signal Processing (DSP) algorithms and are found in phased array antenna systems (for radar and radio astronomy), wireless (baseband) communication (for wireless LAN, digital radio, UMTS [30, 75, 106]), multi-media, medical imaging and sensor networks.

A key characteristic of what is referred to here as a *streaming application* is that it can be modelled as a dataflow graph (DFG) with channels (streams of data items, represented by the edges) between tasks (computational kernels, represented by the vertices) [23]. The qualification "real-time" implies that timeliness is part of correctness. As a consequence, throughput, latency and jitter are *constraints* rather than (optimization) *objectives* [88]. In hard real-time systems no deadline may be missed, as that may lead to dangerous situations. In soft real-time systems, missing a deadline is not catastrophic, but does degrade the system's total performance.

Even though no firm guarantees are given for such systems, the goal is to keep the QoS high. In short, an important property of real-time systems is that nothing is gained by delivering a higher QoS than the application asks for.

For any kind of real-time behaviour (soft or hard), applications need to have predictable behaviour in terms of time and spatial (i.e. hardware) resource usage [16] so that at least some QoS prediction can be made. Predictable behaviour means that execution time and resource usage are bounded. Tighter bounds give better-or-equal predictability. Typical real-world applications that fall into this category display a high degree of regularity in the communication between tasks and have a semi-static life-time [106], i.e. typically in the order of minutes, rather than milliseconds.

1.3.2 SPATIAL RESOURCES: TILED SYSTEMS

As stated in the introduction of this section, embedded systems are commonly built up out of IP blocks. IP blocks of any granularity can be combined into components of different granularity: A system may consist of a single Printed Circuit Board (PCB) or of multiple interconnected PCBs. One PCB can contain many ICs-packages. Every IC-package can contain multiple chips—a so called System in Package (SiP). Every chip may contain many different IP blocks, i.e. may be a Multi-Processor System-on-Chip (MPSoC). MPSoC integration is gaining particular popularity in embedded systems, because of its compactness and energy efficiency (compared to multi-chip solutions).

Recently, considerable numbers of MPSoC designs have been proposed and built. Examples of such MPSoC designs are IBM's Cell [49], Tilera64 [21], Intel's experimental 80-tile [97], Intel's prototype Single-chip Cloud Computer (SCC) [82], Annabelle [85] and the Cutting edge Reconfigurable ICs for Stream Processing (CRISP) project chip [110]. On a more conceptual level, MPSoC design templates have been developed, such as Pleiades [3] and Chameleon [4]. For a more detailed overview, we refer to [4].

What is referred to as a *tiled system* in this thesis, is a multi-processor architecture, where the individual processors can be considered autonomous and composable. Autonomicity means that a processor can be programmed separately from other processors. Separate ALUs or pipelines in a superscalar processor are not considered autonomous. Composability means that a processor can be assigned a task—or tasks already running on the processor can be changed or removed—without directly affecting (unrelated tasks on) other processors. In other words, the QoS of unrelated tasks is not affected, i.e. they still do their jobs correctly and within their guaranteed resource bounds. The same autonomicity must hold for other resources in the tiled system, like memories with a communication assist, Network-on-Chip (NoC) or DMA, I/O modules (A/D converters, etc.), or application specific circuitry. For these (spatial) resources to form one system, they must be interconnected. The combination of an autonomous resource and its interface to the system's intercon-

nect is referred to as a Hardware Element (HwE). Related work often identifies the Processing Element (PE) to be an elementary building block of MPSoCs, which is why the discussion of related work (in chapter 2) uses the term PE. Besides PEs, HwEs also include memories and I/O modules. The rest of this thesis assumes the more general HwE as a basic component. When an MPSoC contains different types of HwEs (i.e. different resources), it is considered heterogeneous.

For the sake of composability, a system's interconnect must also provide QoS guarantees [52]. The NoC paradigm [10, 24], which is gaining popularity in the MPSoC world, has interconnect architectures that provide such guarantees [106], but is by no means the only applicable paradigm. Conventional busses and mixed NoC-and-bus interconnects are all acceptable, as long as their behaviour is predictable, e.g. they can be modelled as latency-rate servers [90]. This is especially relevant when extending systems from MPSoC to SiP and even to multiple chips on a PCB.

1.3.3 On-line resource management

Generally, spatial resource management is the allocation of spatial resources to applications. In the context of tiled systems, spatial resources are HwEs and communication resources. Thus, spatial resource management is the assignment of tasks and channels from the application's task graph to tiles and the interconnect, respectively. The assignment of all tasks and channels of an application is called an *execution layout*. A *feasible* execution layout satisfies the application's QoS constraints. An execution layout's *quality* depends on the extent to which it optimizes resource usage and extra-functional costs like energy consumption. The quality of a spatial resource management *algorithm* depends on the trade-offs of the platform on which it is used, but is typically a combination of response time, all execution layouts' qualities and the success rate of finding execution layouts for applications.

A downside of heterogeneous tiled systems is that even when only a few HwEs are allocated to applications, there may be no more HwEs of the correct type available to execute a specific task of the application being started. When there are different types of HwEs with the same functionality (e.g. different types of processors, memories with different types of communication assists, etc.), the same task can be implemented for different types of HwEs. Having multiple implementations for the same tasks thus increases the flexibility of the resource allocation in a heterogeneous system. Even when an additional implementation of a task is less energy-efficient, the application's overall energy-efficiency might still benefit from its use, when the closest (in terms of the interconnect) available HwE required for the preferred implementation is far away. The same holds for the latency imposed by computation and communication. For sufficiently large systems, communication costs (in terms of latency or energy) might supersede the added computation cost from a less efficient implementation on a nearby HwE.

In our context, the objective of the spatial resource management is to minimize the energy consumption of the entire application: processing, storage (i.e. memory)

and communication. In principle, the spatial resource management is performed only when a new streaming application is started. This does not strictly exclude dynamic structural changes in an application, e.g. when the signal of a wireless broadcast degrades, the control system of a receiver may be specified to start an extra error-correction task. When new tasks are dynamically added to an application, the execution layout of tasks already running is a constraint for the resource management of the new tasks. An important assumption for on-line spatial resource management, though, is that applications are quasi-static, so that the benefit of the flexibility gained outweighs the added cost of the on-line resource management. Furthermore, on-line spatial resource management algorithms must be fast, because start-up time is often bounded by the application as well (e.g. answering a ringing phone).

To be able to perform the resource management of an application, a spatial resource management algorithm needs a model of the hardware platform and, for the application, the task graph with the corresponding QoS constraints and available implementations of the tasks with their resource requirements, energy costs and behavioural bounds. Some performance figures can already be determined at design-time, e.g. the execution time and energy consumption of various implementations of tasks on specific HwE types. However, some figures can only be determined for a running system. This requires simple performance models (simple in the computational sense, since there may be tight constraints on the time required to find the execution layout).

Performing the spatial resource management on-line implies that *fewer* performance figures can be determined at design-time. It is, after all, only known after the resource management on which HwE a task will be executed, which means that inter-task communication parameters (e.g. latency, energy consumption), for example, need to be determined when starting the application. Likewise, it is only known at application start-up which tasks are already running on a HwE. Therefore, the response time of a task is only known after the on-line resource management has taken place. On-line resource managers and schedulers must not just guarantee their own QoS constraints, but also guarantee that the overall constraints of applications are not violated. This requires schedulers to be asynchronous servers with bounds on preemption [16]. However, the on-line choices are restricted to a finite set of implementations, all of which have properties that are determined at design-time.

Whether an application fullfills all its constraints can only be fully checked after its execution layout has been determined. We use a dataflow analysis [33, 104] for this check, which is beyond the scope of this thesis. As previously stated, only an execution layout that lets the application meet its QoS constraints is considered to be feasible.

1.4 Coordination language SNet

In this section, we discuss concepts of SNet, an asynchronous stream coordination language. These concepts are required for asynchronous combinatorial stream programming.

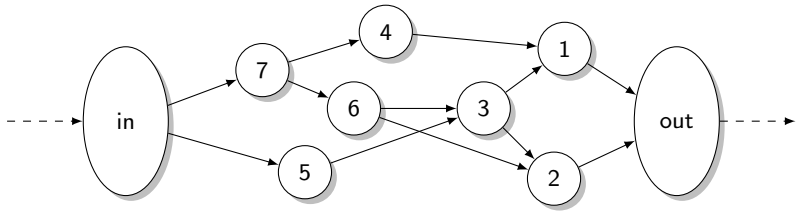
1.4.1 asynchronous combinatorial stream programming

Networked stream programming goes back to Kahn's networks [50] which are fixed graphs with message streams flowing along the edges and stream-processing functions placed at the vertices. The importance of this type of computing is in its simple fixed-point semantics and the static nature of task distribution (discussed above). It is due to these characteristics that networked stream programming is used widely in control systems (for example the Airbus software [13] is written in a stream processing language esterel [11]). However, with the advent of multicore systems and especially large, heterogeneous, many-/multicore architectures, the synchrony found in most programming tools of this kind will become more and more of a limiting factor for throughput and utilization maximization. Consequently asynchronous stream-processing languages, such as SNet [37] are likely to prove to be useful. The principles behind asynchronous stream-processing can be found in [86]; here we only restate some ideas required to understand the work presented in this thesis.

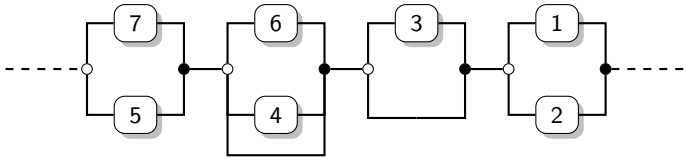
mimo vs. siso

Figure 1.1(a) shows an arbitrary streaming network, where vertices are functions of multiple streams producing multiple streams (the top diagram). This is referred to as mimo. For simplicity, the network is assumed to be acyclic. The input stream α is split by the vertex *In* into streams carrying messages that are intended for specific input ports of individual vertices. The output of the graph is gathered by the vertex *Out* into a single output stream. Assuming that the vertices can respond to the input messages on different ports irrespective of their mutual timing (the assumption of asynchrony), multiple input streams to a vertex can be merged into one, where the messages themselves are labelled with the port information. Similarly multiple output streams could be labelled and merged in such a way. Thus, any asynchronous mimo network can be rewritten to a siso network. The example network is rewritten in figure 1.1(b).

In the rewritten network, the black bullets are non-deterministic stream mergers and the circles are splitters. The position of a vertex in the rewritten network is determined by the longest path to that vertex from the network input in the original network. Bypasses (identity functions) are added when a vertex requires messages from not-immediately-preceding stages. The topology of siso networks can be constructed with algebraic expressions, with networks as operands and

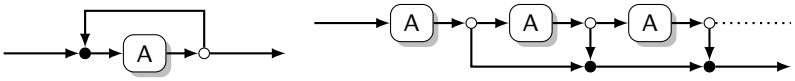


(a) Multiple Input Multiple Output (MIMO) network



(b) Single Input Single Output (SISO) network

FIGURE 1.1 – A MIMO network and its equivalent SISO network



(a) Cyclic network

(b) Unrolled network

FIGURE 1.2 – Unrolling

combinators as operators. Any (valid) expression of this form is again a SISO network. Two combinators are used in figure 1.1(b); serial and parallel composition. SNET provides more than these two combinators; they are described in section 6.2.

Cyclic vs acyclic

Streaming networks are generally cyclic. For synchronous systems with guaranteed resource bounds, cyclic topologies form constraints on resource management [104]. In asynchronous systems, however, cycles in the topology are unbounded cyclic data-dependencies. Cyclic data-dependencies give rice to deadlock and starvation, if resource requirements can not be anticipated. To mitigate the effects of such cyclic dependencies, feedback loops (see figure 1.2(a)) can be converted to infinite

feed-forward topologies (see figure 1.2(b)). This conversion is based on unrolling. For every consecutive visit of a vertex, a separate vertex is instantiated. Instead of feedback loops from a vertex to itself, edges are drawn from a vertex to the vertex representing the next visit. Feed-forward structures expose parallelism, similar to loop unrolling, and thus may be desirable for that reason as well. There is a combinator in SNet for such unrolling feed-forward networks (such as the one shown in figure 1.2(b)).

1.5 Structure of the thesis

This thesis consists of two parts. The first part discusses the work on on-line spatial resource management, coming from an embedded systems perspective. The second part discusses the work on SNet, coming from an HPC perspective. Conclusions and recommendations for future work are combined in the chapter 8 after part two.

Part one is divided over four chapters. Chapter 2 describes the state-of-the-art as discussed in related work. Next, in chapter 3, a formal definition of the problem of on-line spatial resource management and an initial introduction into our solution is given. Chapter 4 discusses our solution and a proof-of-concept implementation of an on-line spatial resource manager. In chapter 5, experimental results for our proof-of-concept implementation are presented and discussed.

Part two contains two chapters. The first, chapter 6, presents a denotational semantics for the language SNet. The second, chapter 7, presents a novel run-time system for SNet.

Part I

Synchronous Dataflow

State-of-the-Art

Abstract – *On-line spatial resource management is a relatively new research area. As such, the state-of-the-art is not yet clearly defined. Therefore, the literature discussed in this chapter covers partial solutions and weakly related areas. We focus especially on solutions for heterogeneous systems. Because the applications suitable for on-line spatial resource management must adhere to strict constraints, some design-related literature is examined as well.*

2.1 Introduction

The detailed design of an application, including the partitioning into communicating tasks and the implementation of those tasks, results in an application specification. Such a specification includes resource budget requirements and QoS constraints. Concurrent programming of applications is a non-trivial practice. It starts with task decomposition. Task decomposition requires not only concurrent programming of an application, but also a sensible grouping of the program into tasks, such that their resource requirements and their performance are predictable and well balanced. This means that the programmer should have a good knowledge of the underlying parallel architecture. After task decomposition, for the resulting task set, resource scheduling, communication specification and synchronization still have to be carried out, all of which may be subject to deadlocks or race conditions. These are responsibilities unique to concurrent design.

Some attempts are made to automate the process of task decomposition of sequential programs, e.g. [70, 98], but manual development is still the ruling paradigm. In general in concurrent design, but especially in resource scheduling, there is a trade-off between multiple, conflicting objectives, like performance levels and costs. Applying *multi-objective optimization* may lead to a set of locally optimal solutions.

In this context, the Pareto set is the set of all solutions that have at least one objective with regards to which they can not be improved without worsening them with regards to another objective [89]. Ykman-Couvreur et al. [107, 108] use design-time exploration to construct a Pareto set that contains multiple implementations of an application. Such design-time exploration of implementation alternatives can also be performed on a per-task basis, after task decomposition. Figure 2.1 shows an example for which each implementation—depicted as a dot—is characterized by a Pareto-optimal combination of performance constraints, resources requirements and costs. The result of this design-time exploration can be used as input for a resource manager.

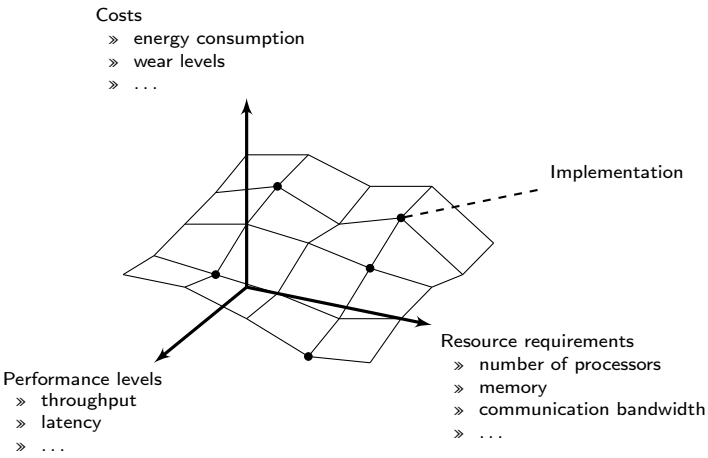


Figure 2.1 – Design-time exploration of the application resulting in various implementations (taken from [107])

2.1.1 Application specification

A streaming application, decomposed into tasks with communication between the tasks, can be represented as a dataflow model. Dataflow is a very common modelling technique for embedded and real-time application engineers [48]. More specifically, Synchronous Data Flow (SDF) graphs [61, 62] are widely used to specify applications, for example in [58]. An entire application can be specified with an SDF graph, where the nodes in the graph represent individual tasks. The edges

between the nodes model the communication channels between the tasks of the application. SDF graphs are a subclass of Petri nets [77, 78]. In this class, firing rules are independent of data values, so that the execution order can be determined at compile time. This ordering allows for a semi-static scheduling strategy, where processor assignment can be determined when starting the application. The use of SDF is discussed in more detail in section 2.4.

2.1.2 Performance guarantees and multi-tasking

Most forms of multitasking require preemptive schedulers. Multitasking systems with non-preemptive schedulers are commonly referred to as cooperative multitasking systems. These systems lack composability, because misbehaving applications can disrupt the QoS of other applications. A non-preemptive scheduler can be used in single-tasking execution environments. Non-preemptive scheduling algorithms are easier to implement than preemptive algorithms and also impose less run-time overhead [47]. Applications in a non-preemptive scheduling context get resources allocated and proceed to run without interruption. Guarantees of QoS can be derived with straightforward analysis when there is no resource contention between otherwise unrelated applications. This assumes that the resources assigned at design-time suffice for the application. Typically, this leads to a low overall system utilization.

To support multi-tasking with real-time constraints, dataflow analysis can be used to derive schedules that guarantee that hard real-time tasks will meet their respective deadlines. Since exhaustive design-time analysis of all possible use-cases is infeasible even for a relatively small application set, run-time scheduling and the implications it has with regards to predictability and performance must be considered. Schedulability analysis takes worst-case waiting times into account, resulting in a very pessimistic result [58, 104]. Kumar et al. compare various analysis techniques in [58]. They show that all the proposed techniques in the multi-processor domain that provide guarantees, have a low utilization. The same work presents a technique that improves utilization, by sacrificing the ability to provide hard real-time performance guarantees. Analogously to the work of Kumar et al., most known solutions that support multi-tasking do not provide hard real-time guarantees.

Wiggers, et al. [103] show that the accuracy of the analysis can be improved by modelling run-time scheduling of shared resources with latency-rate servers [91]. Examples of suitable scheduling algorithms for dataflow analysis are Time Division Multiple Access (TDMA) and round-robin, both of which are latency-rate schedulers [104]. The above authors all recognize the trade-off between increasing scheduler complexity on the one hand and increasing flexibility and utilization on the other, when increasing the allowed amount of resource sharing, e.g. in multi-tasking.

It is common practice in the design of run-time reconfigurable MPSoCs to have a centralized operating system to control the entire MPSoC. Such a system runs on one PE. Other PEs may also run some form of light weight operating system, but the control over the MPSoC as a whole typically resides in the centralized operating system. This includes the system's resource manager. Faruque et al. argue that such centralized Run-time Spatial Resource Management (RSRM) does not scale well into the domain of MPSoCs that consist of hundreds or thousands of PEs [31]. MPSoCs in production today, typically include considerably less PEs, e.g. [21]. Even in the research field, multi-processor chips approaching one hundred PEs are typically homogeneous (to exploit regularity) [97]. Although the scalability concerns raised in [31] are valid, they seem a long time away from being relevant to today's real-life systems.

In [31], Faruque et al. propose a distributed solution that uses a two-step approach. An application is first assigned to a cluster of PEs with sufficient available resources, after which a *cluster agent* solves the original problem in a centralized way, but reduced to the PEs under its management. The work presented in this thesis does not scale to the type of systems dealt with in [31], but could be used as such a cluster agent.

When a new application is started, the RSRM must select suitable implementations for (tasks of) the application. What constitutes the most suitable implementation depends on the current state of the MPSoC, the optimization objectives and QoS constraints to which the application is subject. The RSRM must guarantee that the resource requirements of the application can be fulfilled, before the application is admitted. There is an increasing number of scenarios in which the number of use cases is unconstrained at design-time. For example, any MPSoC that is used as a *user platform*, i.e. the user can download and start an application at any time. For such scenarios, clairvoyant design-time resource management is not possible. At arbitrary time points, applications are added to the MPSoC, as done in [19]. Decisions made by the RSRM for newly started applications may not degrade the performance of applications already running below their QoS constraints. More precisely, a RSRM must adhere [58, 68] to the following conditions:

1. *admission control*: an application is only allowed to start if the system can allocate, upon request, the resource budget required by any of its tasks to meet the application's QoS constraints.
2. *guaranteed resource provisions*: an already running task may never be denied access to its allocated resources by any other task.

If an application can not be added to the system without violating above conditions, the application must be rejected. To resolve such a rejection, either the application's QoS level or the platform state have to be changed. Approaches for dynamic choices of resource requirements of applications are discussed in more detail in

section 2.3.1. The platform state can be changed by stopping running applications (that are considered less critical) or by migrating tasks.

2.2.1 Live task migration

Allowing a resource manager to migrate running tasks from one PE to another further increases flexibility. It improves load-balancing, increases resource allocation success rate and makes resources reclaimable when QoS requirements decrease for a running application. Another advantage is that PEs can be periodically cleared to perform dependability tests [57].

Load-balancing in modern high-end virtualization servers for enterprise systems is based largely on hardware supported live task migration [26]. The granularity of tasks, in this case, however, is considerably larger: Entire virtual machines are migrated at once. The term "seamless migration" is used in this context—e.g. in [95]—to indicate that a user of the virtual machine does not experience performance degradation during migration. QoS guarantees are typically not given. Heterogeneity is typically limited to processors with different accelerators and instruction set extensions around the same core instruction set, e.g. x86 processors with and without SSE3 extensions. Supported processors must offer uniform support for the identification of their capabilities, such as the CPUID instruction requirement in [99].

The application of virtualization techniques in state-of-the-art embedded systems is limited, at best [46]. Providing a uniform abstraction like a migratable virtual machine for a system with different processors of incompatible instruction set architectures is hard. Overall efficiency under such abstractions is typically very low. As an alternative, tasks can be made migratable by external intervention. In order for tasks to be migratable, it must offer *migration points*, at which its state can be extracted and moved to another PE. Nollet et al. [71] demonstrate a heterogeneous system running migratable tasks. Requests for migration may be issued at any time. The time between such a migration request and the moment the task arrives at its migration point is referred to as the *reaction time*. When a migration point is reached, the task's state can be extracted from the PE the task is running on. If the task is migrated to a PE of a different architecture, the run-time system (running on its own PE) can translate the extracted state to a representation suitable for the target PE. The (possibly translated) state is moved to the target PE and the task is started (or resumed) on that PE. The time between the moment the migration point is reached and the moment the task is started on the target PE is referred to as the *freeze time*.

In many cases, it may be possible to bound both the reaction time and the freeze time. Even so, the QoS constraints must be so relaxed and/or the resources so overdimensioned to provide hard real-time guarantees, that this method is considered generally unfit for hard real-time systems. The freeze time degrades the performance of the task being migrated, whereas the reaction time degrades the

waiting time for the task that caused the migration request. The reaction time can be reduced by increasing the number of migration points. However, [71] identify the overhead of checking for pending migration requests at every migration point as the main issue in task migration. They propose a hardware support task migration technique for heterogeneous MPSoCs. Having such a hardware constraint for PEs hinders integration of IP blocks from vendors that do not support (the same) standard. Currently, there is no support for such techniques in compilers and other design-flow tools.

In [109], experiments with the migration of various applications on an ARM architecture are analysed. The total downtime due to migration ranged from 0 to 95 seconds, with an average of 22 seconds. Most of the downtime is consumed by the serialization and deserialization of the task state at a migration point. Zhang, and Pande, describe static compiler analysis methods [109] that optimize the state representation for serialization. The typical performance degradation of tasks compiled with these methods is shown to be around 2%. The range of downtime was reduced by these methods to a worst case of 36 seconds and an average of 8.5 seconds. Even though these performance wins are considerable, downtime is still orders of magnitude out of range for hard real-time embedded systems.

2.3 Subproblems

Four subproblems are commonly identifiable in works dealing with resource management for (embedded) multi-processor systems: partitioning, binding, mapping and routing. Not all authors recognize all of these as relevant. The identification of these subproblems and the recognition that together they describe the entire problem of spatial resource management is a contribution of this thesis.

Works discussing homogeneous architectures do not discuss binding. Many authors consider partitioning and binding as integrated design-time problems. Mapping is considered in all related work, but for some it is a by-product of routing, e.g. [19, 65, 87]. Some authors consider routing to be a trivial problem. In this thesis, partitioning is considered as a design-time problem. Thus, work focussing on partitioning is not discussed here. The remainder of this section discusses related work for the three considered subproblems.

2.3.1 Binding

Binding is the decision on what type of PE to run a task. In approaches where hard- and software are developed simultaneously, binding is a hardware/software co-design problem. Although work on performing just-in-time compilation (specifically, just-in-time (re)targeting) is ongoing, e.g. [59], it is still uncommon in embedded systems. Therefore, binding performed at run-time is constrained primarily by the availability of multiple implementations for a task.

Ykman-Couvreur et al. use a Multi-dimensional Multi-choice Knapsack Problem (MMKP) formulation [107] to obtain a (near-)optimal solution to a binding type of resource allocation problem. However, because an MMKP is NP-hard and large execution times and resource requirements are needed to solve this problem, exact algorithms are not applicable for run-time resource management. Ykman-Couvreur et al. discuss a fast heuristic for solving this MMKP. They reduce their multi-dimensional resource requirements to (scalar) cost and tasks are sorted by their cost. Then, a greedy algorithm selects minimum cost solutions for the tasks in-order. Other works concentrate on a single cost parameter, e.g. energy consumption [31] or execution time [72]. Using this single cost parameter, they also apply greedy selection algorithms.

The only related work found in which systems are explicitly overloaded with regards to the binding is [55]. In this work, Kim et al. evaluate heuristics for scheduling tasks in a heterogeneous environment. They schedule only independent tasks, i.e. without inter-task communication. They try to map the best subset of tasks onto the platform. The lack of inter-task dependencies makes this inapplicable for the type of applications discussed in this thesis.

Carvalho et al. consider binding a design-time problem. In [18], they identify two classes of tasks: hardware tasks and software tasks. They use GPPs for the software tasks and bind the hardware tasks to reconfigurable logic or ASICs. Which tasks are implemented in hardware and which in software is assumed to follow from the application specification in [18].

Nollet et al. [72] combine the binding and mapping of tasks. For every task, Nollet et al. calculate the normalized execution time variance over all supported PEs. Tasks with a high normalized execution time variance are very sensitive to their PE-assignment. Sensitive tasks should preferably be bound first. A task's priority is its execution time variance multiplied by its communication requirements factor. Nollet et al. sort tasks by their priority. PEs are sorted by their load and available communication resources. Tasks are iteratively mapped to the best fitting PEs. In [72], reconfigurable hardware often has the highest preference. However, reconfigurable hardware is scarce in the system described. An ad hoc heuristic is employed as a post-processor to minimize waste of this scarce resource. Because Nollet et al. do not consider non-functional factors (e.g. energy consumption), the resulting binding and mapping may have poor performance with regards to non-functional factors.

2.3.2 Mapping

Mapping is the decision on which PE to run a certain task. Most related work dealing with mapping considers only homogeneous systems. For applications where the binding is fixed, mapping usually seeks to minimize communication costs and fragmentation.

As discussed above, no systems currently exist that allow task migration while still guaranteeing QoS compliant behaviour. Therefore, a mapping is considered fixed for the entire life-time of the application. Consequently, choices made for the mapping of an application impose constraints for the mapping of any application started later. When an application is started on a heavily loaded system, it may be mapped to resources that are far apart from each other. Even if, consecutively, the load on the system is reduced, this application still imposes constraints on the use of a lot of resources. Simple linear allocation methods have been shown to cause heavy fragmentation after a relatively short sequence of allocations [51].

Fragmentation can be split into internal fragmentation and external fragmentation. Internal fragmentation is (the scattering of) unused capacity of resources to which tasks are mapped. External fragmentation is the scattering of free resources among resources to which tasks are mapped. Little explicit attention is paid to fragmentation in the literature. However, external fragmentation is a component in the cost function for mappings in [64]. In [68], algorithms are "adjusted to fill up partially filled PEs" to reduce internal fragmentation.

Mapping heuristics

In some works, the low-complexity First Fit (FF) algorithm is used for the (initial) mapping of tasks to PEs. Moreira et al. [68] support multi-tasking by mapping every task to an intermediate Virtual Tile (VT). Then, each VT is assigned to a PE, taking communication channels into account and allowing multiple VTs to be mapped to a single PE. Moreira et al. describe an architecture with a ring-topology, where every router on the ring is connected to three PEs. As a consequence, bandwidth is relatively scarce compared to the amount of computational power. Tasks with heavy communication between them should preferably be mapped onto PEs connected to the same router. The authors' First Fit with Clustering (FFC) algorithm is specialized for the ring architecture compared with a general FF algorithm. Although the FFC algorithm decreases bandwidth usage, it has an adverse effect on the mapping success rate when the system becomes saturated [68]. Therefore, Moreira et al. use an unclustered version of their algorithm when FFC fails.

Chou and Marculescu [19] apply a node colouring approach to the mapping problem. Initially, all tasks are coloured white. When a suitable candidate PE is found for a task, that task is coloured grey. When a mapping of a task to a PE is chosen and fixed, the task is coloured black. Tasks are grouped by the required PE type and ordered by decreasing communication demands. Iteratively, for the first task from the smallest PE-group, an available PE is sought of the correct type. This task is now grey. If the neighbours of this task (in the task graph) are either grey or black, a minimum distance PE of the correct type is sought, such that the distance to the grey or black neighbours is minimal. Now the task is coloured black.

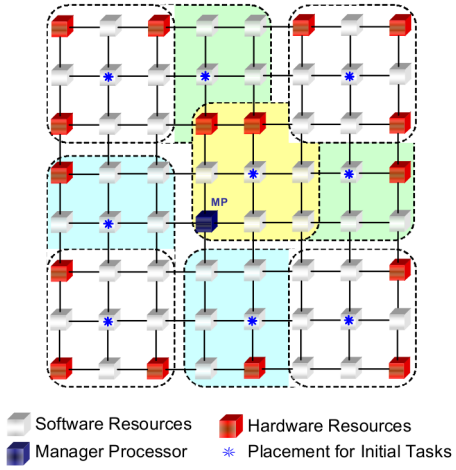


Figure 2.2 – Clustering: Dashed lines denote the cluster limits [18].

Problem reduction: clustering

Carvalho et al. [18] propose a clustering approach to partition the mapping problem. They identify a master task in every application. The other tasks in an application are slave tasks. The master task must issue requests for the mapping of the slave tasks. It is shown that mapping the slave tasks at random can result in undesirable mappings [18]. To promote locality of tasks of the same application, Carvalho et al. define clusters of PEs. These clusters may (partially) overlap. At most one master task is mapped to a cluster. The slave tasks of a master task are all mapped in the same cluster. An example, taken from [18], is shown in figure 2.2.

In this example, a system is divided into nine clusters, each consisting of nine PEs. Where these clusters overlap, the resource management for slave tasks of different applications interacts, i.e. resource sharing is restricted to where clusters overlap. Master tasks communicate with the Manager Processor (MP) to request and release resources. When a master task requests resources for a slave task, a PE must be selected from the master task's cluster. Because resource management is centralized in the MP, there are no race conditions for the PEs that are part of multiple clusters. In this system, slave tasks can be started and stopped any number of times. A strong downside of this approach is that the number of master tasks (and thus the number of applications) is limited by the number of clusters. The authors argue that this constraint is required to prevent application deadlock when the system runs out of resources.

The overall performance is very sensitive to the initial placement. How clusters are chosen and how interactions with the environment occur (e.g. introduction and

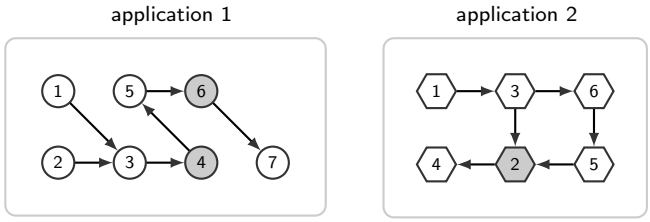
termination of applications) is not described in [18]. Also, some responsibility for bootstrapping the application is moved from the resource manager to the application's master task. This results in less distribution transparency and increased communication overhead, which are both not desirable. Furthermore, QoS must now be guaranteed by the master task.

Faruque et al. [31] also use clustering for resource management in large MPSoCs. They describe a system with task migration capabilities. When an application is started, a cluster negotiation algorithm is run. If no suitable cluster can be found for the application, then task migration requests are issued to free resources in a promising candidate cluster. If this also fails to provide a sufficiently large cluster, a re-clustering algorithm negotiates between the candidate clusters and its neighbouring clusters. It attempts to let these clusters exchange some PEs. When this also fails, the resource manager fails to find a mapping. For resource management inside a cluster, Faruque et al. use the algorithm of Hansson et al. [41]. This algorithm was originally developed for usage at design-time. The combination of a task and a PE is assigned a cost, that is the sum of a four factors. The first is the average distance of the PE to all other PEs in the same cluster. The second is the cumulative bandwidth requirement of the tasks' communication on the PE. The third is the cumulative resource requirement of the tasks on the PE. The fourth is the sum over the Manhattan distances between the already mapped neighbours (in the task graph) of the task and the PE, multiplied per channel by the channel's communication volume. Unmapped tasks are sorted, first, by availability of PE types and, second, by their bandwidth requirements. Iteratively, the lowest cost pair of task and PE is chosen as a mapping. All other candidate pairs for the same task are discarded. Faruque et al. [31] claim a lower complexity for their algorithm than that of Carvalho et al. However, they do not give results in terms of performance measures.

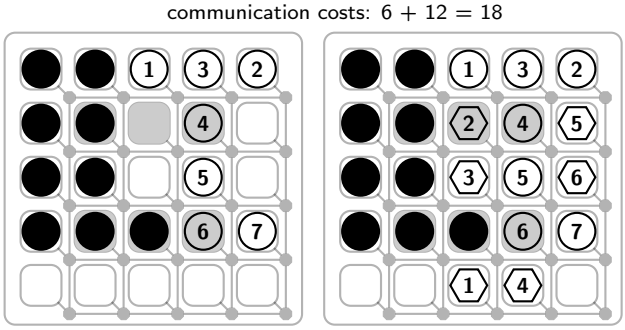
Problem reduction: region selection

Chou and Marculescu [19] describe a mapping approach using region selection. Their approach requires two cost factors: the dispersion factor and the centrifugal factor. The *dispersion factor* of a PE is defined as the number of its neighbours that are idling. The *centrifugal factor* of a PE is defined as the Manhattan distance between it and any PE at the border of the region occupied by the application being mapped. The cost associated with a PE is the sum of the dispersion factor and the centrifugal factor. After the first task is mapped to a PE, a region is constructed by selecting those PEs around that first task's PE that have the lowest cost. A simple scenario is illustrated in figure 2.3 (taken from [19]). It shows that fragmentation is significant after mapping the two applications (figure 2.3(a)) using a greedy algorithm (figure 2.3(b)). Using region selection (figure 2.3(c)) the fragmentation is significantly reduced.

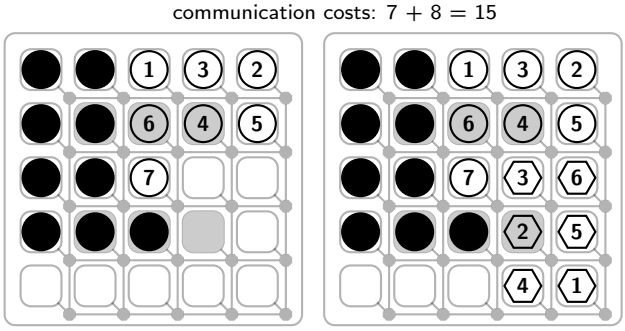
Region selection steered mapping prefers PEs that have a high probability of becoming isolated. This is due to the dispersion factor. When most of a PE's neighbours



(a) Two applications that have to be mapped



(b) Greedy mapping algorithm



(c) Region selection steered mapping

Figure 2.3 – An example of two approaches in incremental application mapping on a heterogeneous MPSoC, where both the degree of fragmentation and communication costs play a role. Black circles indicate tasks already active. Shading of PEs indicates their type. Tasks are shaded according to the required PE type. Communication costs are expressed in terms of distance.

have tasks mapped onto them, that PE has a low dispersion factor. The centrifugal factor adds preference for application mappings with high locality, i.e. when tasks are mapped closer together, the centrifugal factor is lower.

After a region has been selected with sufficient PEs to map the application, the resource manager must map the tasks to the PEs in the region. Effectively, region selection has reduced the size of the mapping problem, but the principle problem remains the same. In [19], computational experiments show that region selection is able to reduce inter-task communication overhead with 25%, but these results are biased towards dense task graphs. Sparse task graphs win considerably less from reducing the total distance between all tasks, i.e. the span of the selected region. Unfortunately, the authors do not evaluate the mapping success rate, which we expect to increase when the degree of fragmentation is lowered.

2.3.3 ROUTING

For systems with network-based interconnects between PEs, inter-task communication must be routed through the interconnect. For routing on the scale of systems considered in this thesis and the related work, there are many algorithms available. Execution times of these algorithms for networks with tens or a few hundred nodes are not significantly different [54]. The choice for a good routing algorithm depends mostly on the required capabilities (whether edges are weighted, whether weights can be negative, whether a strong heuristic can be derived from the network structure) [28, 32, 45]. However, the constraint that communication resources must have predictable behaviour [52, 106] and resource management must guarantee QoS is relatively new. A popular form of on-chip interconnect is the NoC [10].

A NoC has a limited number of physical links between each pair of routers. How the finite communication capacity offered by the physical links is shared among different channels depends on the router architecture. In embedded systems, however, time-sharing the physical links is a popular approach. One way to implement time-sharing is by performing arbitration on the output of a router [52, 106]. Another way is by implementing static schedules [35]. Time slots on these physical links are often grouped into so called virtual channels [54]. Per inter-task communication channel, a resource manager can allocate one or more virtual channel(s) through every physical link and router on a route from a source PE to a target PE. Depending on the router architecture, assigning more virtual channels may make more bandwidth available, or give the corresponding channel a higher priority, lowering the communication latency. Most NoC architectures offer only a limited number of virtual channels per physical link. This means that the number of connections (regardless of e.g. bandwidth requirements) is usually limited.

Commonly, NoC architectures guarantee a lower bound on throughput and most offer an upper bound on latency. Guaranteed throughput typically comes at the expense of very low utilization, although research to improve utilization under guaranteed throughput is a topic of research (e.g. [66]). Guarantees are generally

provided from the router upwards. It is often hard to derive from local guarantees, properties of the behaviour of the network as a whole. Exploration of NoC behaviour is a time consuming simulation process. We have contributed some work to the field of accurate and fast simulation of large NoC (based) systems, that are not discussed in this thesis, but rather referred to as related work [PhH:10, 11, 12] .

Resource managers must be aware of the capabilities and limitations of the NoC architectures under their control. Routing communication channels depends strongly on previously routed channels, i.e. applications already running. When the capacity of a physical link between two routers is completely consumed, an alternative, usually longer route may work around this network congestion. This is a suitable solution to fulfil the bandwidth requirements. However, latency constraints are very sensitive to longer routes. Therefore, latency constraints are often given a higher priority than bandwidth constraints. Srinivasan and Chatha [87] have a priority based policy in their solution for design-time routing on NoC architectures. They guarantee that channels with tight latency constraints are given priority over channels that may have high bandwidth requirements, but that have very relaxed latency constraints. Every communication link c that must be routed is ordered to descending priorities:

$$\rho(c) = \frac{bandwidth(c)}{latency(c)^k}$$

The integer k is determined, such that the communication link with the highest bandwidth requirement, c_i, has a lower priority than the link with the tightest latency constraint, c_j:

$$\frac{bandwidth(c_i)}{latency(c_i)^k} \leq \frac{bandwidth(c_j)}{latency(c_j)^k}$$

Alternatively, Chou and Marculescu [19] define an architecture that has separate networks for data and control. These networks are separated to ensure that data transmission does not interfere with the control messages generated by the Operating System (OS), because in their applications, control messages are more sensitive to latency. In their resource management, routing is not considered, because tasks are only mapped to PEs when the bandwidth requirements are met. Chou and Marculescu do not explain how this check is performed.

In [18], the utilization of the NoC is measured using mapping heuristics. The NoC topology, the routing algorithm and the communication delays are modelled, to compare different heuristics. The best performing mapping heuristic is "Path Load", which minimizes the requirement for the links of the NoC for each new task inserted into the system.

Moreira et al. [68] use a NoC architecture that requires static schedules for every router. They use a network graph model that models time slots as independent vertices. To model one router, as many vertices are added as there are time slots

in that router. To route from one time slot to the next in the same router, implies that data must be stored for one cycle. This means that the router's buffer capacity forms a constraint to the maximum number of transitions from one time slot to the next in the same router. When a route passes from one time slot in a router to the next time slot in another router, the single-cycle communication delay is modelled. This approach reduces path finding and slot allocation to a single problem, at the cost of a larger and denser network graph. The problem of finding several paths in this network graph is known as the *Directed Edge-Disjoint Paths Problem*, which is known to be NP-complete [39]. Because of this prohibitive complexity, Moreira et al. [68] route channels incrementally instead. What percentage of the bandwidth can be reserved depends on the control overhead introduced by the resource manager.

2.4 Validation

Even if enough communication resources are allocated to the application, the question remains whether the application performs well enough. The QoS of an application, given a binding, mapping and routing, must be validated. Validation is not required for applications that do not have any performance constraints.

In [58, 68], an application is specified as an SDF graph. Kumar et al. [58] evaluate the performance of an application mapped to a MPSoC platform. A combined model of the platform and the application can generate performance figures, such as throughput and processor utilization. These results are obtained at design-time, and are used to create a resource manager. The resource manager then monitors the applications at run-time, and intervenes to ensure that all the application are able to meet their throughput requirements. It is, however, not guaranteed that every constraint is honoured at all times.

All related work on validation methods is still used at design-time. In chapter 4 we discuss some related work that we use to realize the validation phase at run-time. That approach uses SDF graphs as well.

2.5 Optimization criteria

In streaming applications, the QoS level is a given constraint. Nothing is gained by optimizing the resource management to provide a better QoS level than required. Optimization of resource management consequently involves optimizing the run-time of the resource management itself and/or properties of resource management other than QoS.

For comparison of proposed algorithms, performance measures are needed. In some works [31, 107], computational effort is regarded as a performance measure. However, we consider algorithm complexity a design constraint like in [87], rather

than an optimization criterion. A meaningful constraint on the start-up delay of an application allows at least ten milliseconds, as this is the order of start-up delays in a Linux OS [19, 107]. This is due to the interrupt timer being set to 100 Hz. Depending on the target platform, this tight constraint may be relaxed. An upper bound on the start-up delay is that a solution must be computed faster than the anticipated average arrival rate of new applications [55]. As this period is relatively long in embedded systems, it is not a realistic estimation of the acceptable delay that may be introduced by the resource manager. We claim that no universal constraint can be formulated in terms of start-up delay.

If it is possible to fulfil the request to execute a certain application, the system has to allocate the resources to that application, no matter what costs are involved. At the point where the system becomes saturated, both the availability of resources and the fragmentation degree of resources determines whether a next application is still mappable. Benchmarking the algorithm with numerous scenarios, as done in [68, 72, 87], gives an estimation of the *mapping success rate*. We think that this rate is the most important factor for measuring the quality of solutions. Given a constant success rate, we can optimize the solution towards other performance measures, e.g. *energy consumption* or *resource usage*.

We mentioned in this chapter that energy consumption is the main motivation of heterogeneous platforms. However, estimating energy consumption is very complex, due to the many relevant factors [8]. This is illustrated by the fact that all works that consider energy consumption use different energy consumption models. For example, Wolkotte et al. present in [105] some energy models that compare two NoC architectures with a shared-bus architecture. The derivation of those models shows that the final equation is highly dependent on the architecture and the technology of implementation. The advantage of such an equation, is that once it is derived, it is rather simple to use. However, if we do not want to know absolute energy consumption values, we can simplify the energy model in [105] by stating that less hops in the NoC cost less energy. Unfortunately, for the most important decision that takes energy consumption into account, the binding decision, it is unclear what energy model could be used as an optimization criterion.

2.6 Conclusion

It is hard to compare the various approaches described in this chapter in qualitative terms. We think that there is a need for a generally applicable algorithm, that can be fine-tuned by specific objective functions. In chapter 4, we describe heuristics for each resource management subproblem. At various points, these heuristics can be instructed to optimize towards some objective, such as high performance or low energy consumption.

On-line spatial resource management

Abstract – *A formal definition of what constitutes* on-line spatial resource management *is given in this chapter. To this end, all platform hardware and application software is modelled and the minimum requirements of both are made explicit. The concept of* execution layouts*—the results of on-line spatial resource managers—is introduced. Qualitative criteria are discussed, by which resource managers and execution layouts can be compared. Finally, the heuristic approach described and evaluated in more detail in the following chapters is introduced and related to the formal definition.*

3.1 Structural Definitions

In this section, we describe the *structure* that combines individual components into a system. Only the relations between components are taken into account here, as opposed to their state. The latter is discussed in section 3.2.

3.1.1 hardware platform

As mentioned in the introduction of this thesis, there are many kinds of spatially partitioned systems. In the domain of embedded architectures, the most common

Parts and earlier revisions of this chapter have been published in [PhH:2, 3, 4].

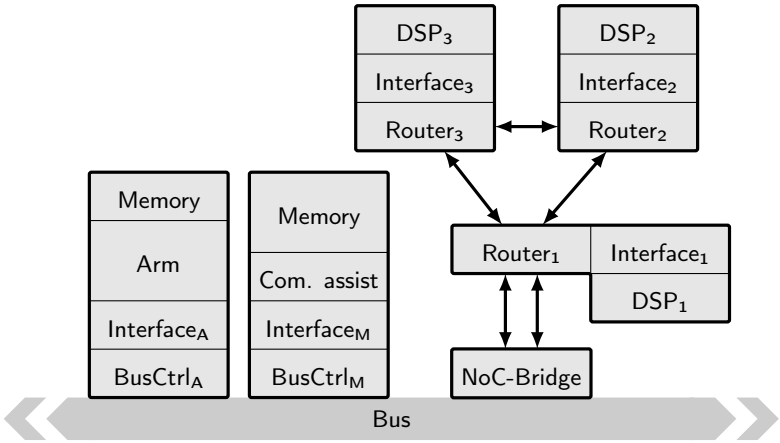


Figure 3.1 – The hypergraph representation of the example architecture.

occurrance of such systems are MPSoC. In this section, we give a formal model of such hardware systems, which we shall refer to as *platforms*. The model is kept sufficiently general to cover a large class of platforms; those that can be managed by on-line spatial resource managers.

Any resource that can perform a (e.g. computational) task is called a HwE. The class of components grouped in the term HwE is a larger class than the more commonly used PE (see chapter 2). HwEs also include, for example, memory and I/O modules. Communication between HwEs is facilitated by the platform's interconnect. This interconnect consists of *routers*, connected to each other by *links*. Even though the terminology stems from the NoC paradigm, interconnects are by no means restricted to NoCs. The term 'router' is used to denote any kind of interconnection element that controls the direction in which data flows, e.g. NoC-routers, NoC-to-bus bridges, bus interfaces, etc.

To clarify notation, consider the example architecture depicted in figure 3.1. This platform is heterogeneous with regards to HwEs, routers *and* links. The routers and the bidirectional connections between them, together form a NoC. Therefore, a NoC-bridge is required to connect the bus and the NoC. The bus controllers, NoC-bridge and routers depicted, are all modelled as routers, since they all influence the direction of data streams in the interconnect. The bus shown allows communication between all connected components, but all communication shares the bus as a single resource. Therefore, the bus should be represented by a single object—in this case, a single link—in our formal description. The same single object representation requirement holds for the bidirectional connections depicted in the NoC between the DSPs. On the other hand, both connections between the NoC-bridge and the NoC are unidirectional. Thus, the representation of links

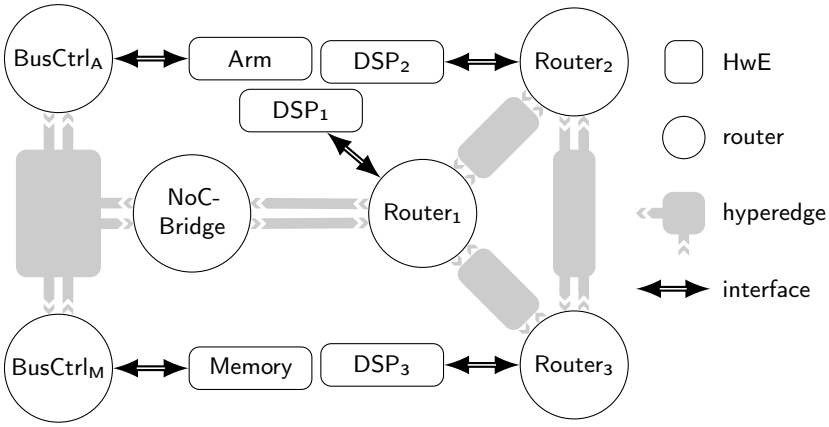


Figure 3.2 – The hypergraph representation of the example architecture.

should reflect direction. Because of these requirements, we use *hypergraphs* to model interconnection. Hypergraphs are a generalization of graphs, where edges (referred to has *hyperedges*) can connect more than two vertices. A hypergraph is *undirected*, if hyperedges are described by a set of vertices. A hypergraph is *directed*, if hyperedges are described by pairs of sets of vertices, viz. a 'from' set and a 'to' set.

Formally, an *interconnect* is represented by a directed hypergraph $N = \langle R, L \rangle$, where R is a set of *routers* (vertices) and $L \subseteq \mathbb{P}R \times \mathbb{P}R$ (where $\mathbb{P}$ denotes the powerset) is a set of *links* (directed hyperedges) between routers. Links are defined as hyperedges, so that both busses and bi-directional point-to-point links can be expressed in this model, besides 'simple' unidirectional point-to-point links.

To further describe a platform, let E be a set of HwEs and $F \subseteq E \times R$ be a set of *interfaces* of HwEs with routers. Interfaces allow bidirectional communication. A *platform* P now is a quadruple $\langle E, R, F, L \rangle$, which, again, is a hypergraph: $\langle E \cup R, F \cup L \rangle$. Note that, following from this definition, there are no direct connections between HwEs. Furthermore, we assume that P is *weakly connected*, i.e. there are no unconnected subgraphs.

The example architecture given in figure 3.1 can now be modelled in the way described above. The resulting hypergraph is depicted in figure 3.2. Names in the model must be unique, therefore the bus controllers of the memory and the Arm were renamed (to $\mathrm{BC_M}$ and $\mathrm{BC_A}$, respectively), and the routers of the NoC, the DSPs and their interfaces (names not depicted in figure 3.2) are numbered. The hyperedges, representing links between routers, are shown in grey. Interfaces are shown as bidirectional double arrows. This figure shows the difference between busses, bidirectional links (between the routers) and unidirectional links (between the NoC-bridge and router one).

3.1.2 SOFTWARE APPLICATIONS

An *application* is represented by a directed graph $A = \langle T, C \rangle$, where T is a set of *tasks* and $C \subseteq T \times T$ a set of *channels* between tasks along which tasks communicate with each other. Tasks are functions mapping input streams to output streams. Implementations are realisations of tasks. In other words, implementations are executable units that compute the function of their corresponding task, i.e. for any task $t \in T$ and a corresponding implementation $i \in I$, the semantics of i is t.

Several implementations may exist for a task. The compatibility of implementations and HwEs is guaranteed by their *types*. We say that implementation i and HwE e have the same type, denoted $\tau(i) = \tau(e)$, if and only if i can be executed on e given sufficient resources are available. For example, when program code is compiled for a specific instruction-set processor, the resulting binary—in this context, this is precisely one implementation—can only be executed on HwE that support the instruction-set the binary requires. Because the platform may contain HwEs of different types, adding implementations for a task gives more flexibility to the online spatial resource manager. For a task t, there need not be implementations for every type of HwE. The set of implementations for task t is denoted by $I(t)$. The subset $I_\tau(t) \subseteq I(t)$ denotes the set of implementations of t that can be allocated to HwEs of type τ.

3.1.3 PATHS

To run an application, for every task one implementation has to be chosen and this implementation has to be mapped onto a single HwE. If two tasks are connected by a channel, the channel must be mapped to a sequence of components of the interconnect, that allow a communication between those HwEs, onto which the two tasks are mapped. We introduce an abstraction of the interconnect to *paths*, so that every channel from an application can be mapped to a single path. This abstraction leads to a 'higher order graph,' in which edges *are* the paths in the platform.

Let $L^\star$ be the set of all cycle-free paths[1] over a platform $P = \langle E, R, F, L \rangle$. Because paths connect HwEs, a path starts and ends with an interface, connecting HwEs to routers. The number of routers on a path is arbitrary (albeit ≥ 1), but between every two consecutive routers there has to be a link. A path is only considered *valid* if every consecutive pair of components (interface, router or link) is connected in P. As a result, we get a *pathed platform* $P^\star = \langle E, L^\star \rangle$, which is a directed multi-graph (a graph that may have several edges between a pair of vertices).

[1] In a hypergraph, a path is described as an alternating sequence of vertices and edges. It is cycle-free, if no two vertices occur twice in it.

3.1.4 Execution layout

If a software application has to run on a platform, we have to associate HwEs to tasks and paths to channels. Clearly, this has to be done in such a way that the necessary implementations exist and that the capacity of the platform is not exceeded.

An *assignment function* $\mathfrak{A}$ is a function, which maps an application A to a pathed platform $P^\star$. More precisely, for every task $t \in T$, $\mathfrak{A}(t)$ is an HwE in E and for each channel $\langle q, r\rangle \in C$, $\mathfrak{A}\langle q, r\rangle$ is an edge from $\mathfrak{A}(q)$ to $\mathfrak{A}(r)$ in $L^\star$. Where required, the part of $\mathfrak{A}$ that maps tasks onto HwEs is referred to as $\mathfrak{A}_\pi$ and the part of $\mathfrak{A}$ that maps channels onto paths is referred to as $\mathfrak{A}_\gamma$.

An *implementation selector* $\mathfrak{I} : T \to I$ is a function which projects tasks onto implementations. An *execution layout* $\mathfrak{L}$ can now be defined as a tuple $\langle \mathfrak{A}, \mathfrak{I}\rangle$ of a task assignment function $\mathfrak{A}$ and an implementation selector $\mathfrak{I}$.

If a task t is assigned to an HwE of type τ, i.e. $\mathfrak{A}(t)$ is of type τ, an implementation of t for an HwE of type τ should exist and the implementation selector must select such implementation. If this holds for all tasks in an execution layout, this execution layout is considered *adequate*. In other words, an execution layout $\mathfrak{L} = \langle \mathfrak{A}, \mathfrak{I}\rangle$ is called adequate, if and only if for every task $t \in T$ it holds that $\mathfrak{I}(t) \in I_\tau(t)$, where τ is the type of $\mathfrak{A}(t)$.

3.2 Resources: Capacities & Requirements

All HwEs in platform P represent resources with finite capacities. One can think of computational and memory capacities, but also of the maximum number of tasks that can be assigned to it; e.g. ASICs can not switch between tasks, so they have a maximum of one task, while an ARM may be able to serve as many tasks as there are slots in its TDMA scheduler. Next to the HwEs, the interconnect is also a resource with finite capacities. Examples of such capacities for the interconnect are link bandwidth, router TDMA slots, number of virtual channels etc.

Thus, relevant (local) capacities of a platform can be expressed by *capacity vectors*. Let $\mathcal{C}_E(e), \mathcal{C}_R(r), \mathcal{C}_L(l)$ and $\mathcal{C}_F(f)$ denote the capacity vectors of an HwE e, router r, link l and interface f, respectively. All capacity vectors for the same kind of elements (HwEs, routers, links and interfaces) are considered to be of the same 'shape,' i.e. every capacity vector of any HwE always has the same components[2]. For simplicity, we assume vectors of independent components.

As an example, consider again the platform from figure 3.1 modeled in figure 3.2. We now fill in the resource capacities of this platform. The numbers in this example have been chosen arbitrarily, but do originate from real-world platforms. First of all, the interfaces of this platform have so called 'communication assists' that serve

[2] In this context, 'component' refers to *vector components*, not to *hardware components*.

	Comm. Assist locks (CAL)	Virtual Channels (VC)	Bandwidth (BW)	Scheduler budget (SB)	Guarranteed Throughput streams (GT)	Best Effort streams (BE)
Interface_1	2	3	1000	·	·	·
Interface_2	3	3	800	·	·	·
Router_1	·	·	·	20	3	4
Router_2	·	·	·	15	3	5
Link_{1-2}	·	4	1600	20	·	·

Table 3.1 – Resource vector examples

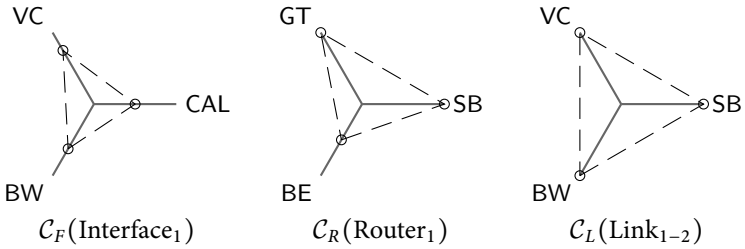


Figure 3.3 – Resource vector examples (graphical representation)

as arbiters for the processing HwEs's local memories and other on-HwE resources. The number of locks for concurrent resource access in these communication assists is limited, so the number of locks is a resource in interface capacity vectors. Next, interfaces have limited bandwidth and a limited routing table, so they can only service a limited number of virtual channels. Arbitration for links and routers is performed with budget schedulers [104] and so the scheduler budget for any channel mapped onto them is limited. Finally, routers distinguish two types of virtual channels: guaranteed throughput and best effort. These are modeled as separate finite resource capacities. Table 3.1 shows the capacities of all the interconnection components.

Capacity and requirement vectors can be represented graphically by radar charts. The axes represent components from the vectors. Vector components that are irrelevent to a type of HwE or interconnection component are omitted (i.e. since routers in this example have the components SB, GT and BE, any $\mathcal{C}_R$ vector is depicted with these axes only). Figure 3.3 shows some vectors from the example above.

As a dual to the notion of capacities of the hardware, software applications have resource requirements, described by *requirement vectors*. For task t, every implementation $i \in I(t)$ has a requirement vector $\mathcal{R}_I(i)$. Similarly, every channel c has a

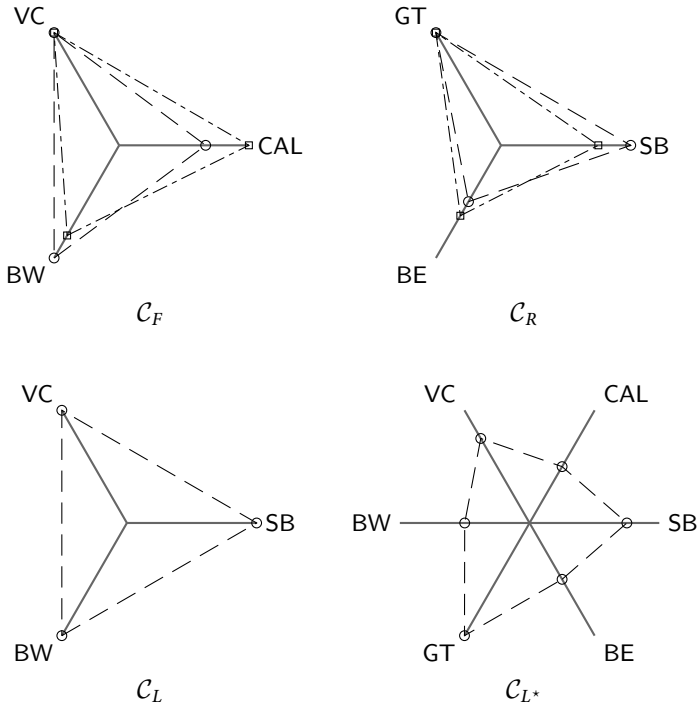


Figure 3.4 – Resource vector compositing example (graphical representation)

requirement vector $\mathcal{R}_C(c)$.

3.2.1 Composited and adjusted capacities

For the description of the proposed methods, it is helpful to have the notion of 'capacity of a path $p \in L^\star$'. We denote such a capacity vector by $\mathcal{C}_{L^\star}(p)$, where all relevant capacities of routers, links and interfaces have an own component in this vector. For a given path $p \in L^\star$, each component in $\mathcal{C}_{L^\star}(p)$ represents the minimum value for this component in the relevant elements of path p, e.g. bandwidth in bits per second.

The example vectors given above can be used to illustrate composited vectors. Consider the path from DSP_1, via routers one and two, to DSP_2. The full path consists of the Interface_1, Router 1, the Link between Router 1 and Router 2, Router 2 and the Interface_2, respectively. The resulting vector is constructed graphically in figure 3.4.

The dual of a capacity vector of a path is the requirement vector of a channel. The vectors in $\mathcal{R}_C$ have the same dimension as those in $\mathcal{C}_{L^\star}$.

When an application is mapped to a platform, obviously, the resources assigned to that application will no longer be available for the next application to be mapped. This platform state is reflected in the capacity vectors. In other words, the capacity vectors reflect the *currently available capacity* of the platform, rather than the capacity of a platform after boot. On the other hand, when applications are stopped, the resources assigned to them are added to the capacity vectors again.

3.2.2 Cumulative requirements

The definitions so far only relate individual implementations and channels to requirements. No definitions have yet been given to express the cumulative resource requirements of a mapped application. For these definitions, the inverse of a task assignment function $\mathfrak{A}$ is required. This inverse is defined in two parts: one part for the assignment of tasks to HwEs and one part for the assignment of channels to paths.

The inverse of the task assignment function with regards to tasks is defined as a function from HwEs to sets of tasks:

$$\mathfrak{A}_\pi{}^{-1}(e) = \{t \in T | \, \mathfrak{A}_\pi(t) = e\}$$

Using this inverse, the cumulative requirement $\mathcal{S}(e)$ imposed on HwE e by execution layout $\mathfrak{L} = \langle \mathfrak{A}, \mathfrak{I} \rangle$ can be expressed as

$$\mathcal{S}_\pi(e) = \sum_{t \in \mathfrak{A}_\pi{}^{-1}(e)} \mathcal{R}_I\left(I(t)\right)$$

Analogously, the inverse of the task assignment function with regards to channels can be defined as a function from paths to channels. However, since routers, links and interfaces may occur in multiple paths, the cumulative requirement on a single router may exceed that on any single path. Therefore, we define the cumulative requirement $\mathcal{S}_\gamma$ as a function from either routers, links or interfaces, to their respective cumulative requirement. To this end, we still require the inverse of the task assignment function from channels to paths, viz.

$$\mathfrak{A}_\gamma{}^{-1}(p) = \left\{c \in C \middle| \, \mathfrak{A}_\gamma(c) = p\right\}$$

so that we can define the set of channels mapped to x (which may be a router, link or interface) as

$$C_x = \left\{c \mid p \in L^\star, x \in p, c \in \mathfrak{A}_\gamma{}^{-1}(p)\right\}$$

and finally cumulative requirement $\mathcal{S}_\gamma(x)$ imposed on x by execution layout $\mathfrak{L}$ as

$$\mathcal{S}_\gamma(x) = \sum_{c \in C_x} \mathcal{R}_C(c)$$

With this notion of cumulative requirement, execution layouts can be checked against the capacity vectors of a platform, to see whether no capacities are exceeded.

An execution layout $\mathfrak{L} = \langle \mathfrak{A}, \mathfrak{I} \rangle$ of application $A = \langle T, C \rangle$ to the pathed platform $P^\star = \langle E, L^\star \rangle$ is called *adherent* if the following constraints are fulfilled:

$$\begin{aligned} \mathfrak{L} \quad & \text{is adequate} \\ \forall e : E \quad & (\mathcal{S}_\pi(e) \leq \mathcal{C}_E(e)) \\ \forall r : R \quad & (\mathcal{S}_\gamma(r) \leq \mathcal{C}_R(r)) \\ \forall l : L \quad & (\mathcal{S}_\gamma(l) \leq \mathcal{C}_L(l)) \\ \forall f : F \quad & (\mathcal{S}_\gamma(f) \leq \mathcal{C}_F(f)) \end{aligned}$$

3.2.3 Minimum capacities

Very few real-world examples exist in which all resources of, for example, a HwE can be simultaneously assigned. In terms of the above, the cumulative requirement imposed on a HwE is hardly ever precisely equal to its capacity. This means that even though a HwE, router, interface or link might still have some remaining capacity, but still be unavailable in any practical scenario. To eliminate these resources from consideration, we introduce a threshold on the capacities, by means of a *minimum capacity vector* $\mathcal{C}^{\min}$ of any resource. What this means is that when the capacity of a resource is depleted below the minimum capacity vector, it is considered generally unavailable. As an example, when HwE e has some resource (e.g. memory) that is completely assigned to running tasks, it holds that $\mathcal{C}_E(e) - \mathcal{S}_\pi(e) < \mathcal{C}_E^{\min}(e)$, in which case it is considered unavailable.

3.3 Constraints and cost

Many applications must deliver at least a specified QoS. The requirements with regards to QoS are expressed in terms of maximum latency and minimum throughput constraints. A constraint is expressed as a relation between two tasks, t_1 and t_2, in the application's task graph. For constraints to be (potentially) binding, a path must exist in the task graph from t_1 to t_2.

QoS constraints of application $A = \langle T, C \rangle$ are specified in two sets for latency and throughput, respectively: $Q_l, Q_t \subseteq T \times T \times \mathbb{R}$ giving upper bounds on latency and lower bounds on throughput, respectively, between pairs of tasks. An execution layout is called *feasible*, if and only if it is adherent and all the application's QoS constraints are met. How this can be proven is discussed in more detail in section 4.4.

Adequacy, adherence and feasibility are all *qualitative* metrics by which execution layouts can be compared. The *quantitative* metric by which (feasible) execution layouts can be compared is *cost*. To this end, let V be a function that assigns a cost (i.e. valuation) to an execution layout. The cost of an execution layout is considered to be the summed total of the cost of all its components, thus, for execution layout

$\mathfrak{L} = \langle \mathfrak{I}, \mathfrak{A} \rangle$, the cost function V is:

$$V(\mathfrak{L}) = V(\mathfrak{I}) + V(\mathfrak{A}_\pi) + V(\mathfrak{A}_\gamma)$$

where the cost of the individual components can be broken down further in the cost of individual assignments, viz.:

$$\begin{aligned} V(\mathfrak{I}) &= \sum_t V(t, \mathfrak{I}(t)) \\ V(\mathfrak{A}_\pi) &= \sum_t V(t, \mathfrak{A}_\pi(t)) \\ V(\mathfrak{A}_\gamma) &= \sum_c V(c, \mathfrak{A}_\gamma(c)) \end{aligned}$$

The details of the cost function are left deliberately undefined. We will return to the considerations for the cost model in chapter 4, where it will become apparent that many considerations for cost modelling are platform dependent. This is also the reason for the liberty taken with the syntax above. For the remainder of this chapter, the existence of a cost function that meets the above criteria suffices.

3.4 Proposed heuristic approach

The previous sections of this chapter have defined a formal framework for on-line spatial resource management. Since it is our goal to determine feasible execution layouts on-line, fast methods are required. Exhaustive search for *optimal* execution layouts is untenable. In this section, we demonstrate this by showing that finding only a task assignment (*given* implementation selector $\mathfrak{I}$) for an execution layout in a rather simple setting is already NP-hard. Thus, heuristics are required to solve this problem. To show how the proposed heuristic—discussed in detail and evaluated in the following chapters—conforms to this formal framework, this section gives a birds-eye view of it.

3.4.1 Complexity as motivation

When we take into consideration only those homogeneous platforms that consist of non-multi-tasking HwEs (i.e. all HwEs may be assigned at most one task and all HwEs are of the same type), we find a relaxed Assignment Problem (AP) [67]. It is relaxed in the sense that there may be more HwEs than tasks. In other words, the mapping of tasks to a homogeneous platform of non-multi-tasking HwEs can be

formulated as the linear program:

$$\begin{aligned}
\min \quad & \sum_{t \in T} \sum_{e \in E} x_{te} V(t,e) \\
\text{s.t.} \quad & \\
& \sum_{t \in T} x_{te} = 1 && \forall e \in E \\
& \sum_{e \in E} x_{te} \leq 1 && \forall t \in T \\
\text{where} \quad & \\
& x_{te} \begin{cases} 0 & \text{if } e \neq \mathfrak{A}(t) \\ 1 & \text{if } e = \mathfrak{A}(t) \end{cases}
\end{aligned}$$

This problem can be extended to multi-tasking HwEs, which introduces constraints on the capacity, making it a Generalized Assignment Problem (GAP) (specifically MINGAP, because the objective is minimisation) [67]. The resulting integer linear program is:

$$\begin{aligned}
\min \quad & \sum_{t \in T} \sum_{e \in E} x_{te} V(t,e) \\
\text{s.t.} \quad & \\
& \sum_{t \in T} x_{te} = 1 && \forall e \in E \\
& \sum_{t \in T} x_{te} \mathcal{R}_I(\mathfrak{I}(t)) \leq \mathcal{C}_E(e) && \forall e \in E \\
\text{where} \quad & \\
& x_{te} \begin{cases} 0 & \text{if } e \neq \mathfrak{A}(t) \\ 1 & \text{if } e = \mathfrak{A}(t) \end{cases}
\end{aligned}$$

The GAP has been shown to be NP-hard and even APX-hard to approximation [73]. Considering that $\mathcal{C}$ and $\mathcal{R}$ are not scalars, but *vectors*, this is a Vector Assignment Problem (VAP). The above clearly demonstrates the need for heuristic approximation, especially if *complete* execution layouts are to be determined on-line.

3.4.2 HIERARCHICAL SEARCH

Considering the prohibitive complexity of exhaustive search, we propose an application domain-aware heuristic: hierarchical search with iterative refinement. We divide the search process in steps, starting with a very coarse grained perspective in the first step and gradually adding more detail. At each step, decisions are made that shrink the search space in the next step. Decisions made in previous steps are considered fixed in later steps.

As is to be expected of heuristics, this abstraction carries with it the danger that decisions made in early steps, using very high-level abstract information, lead to search-spaces in later steps that contain no feasible solutions. Since this only comes

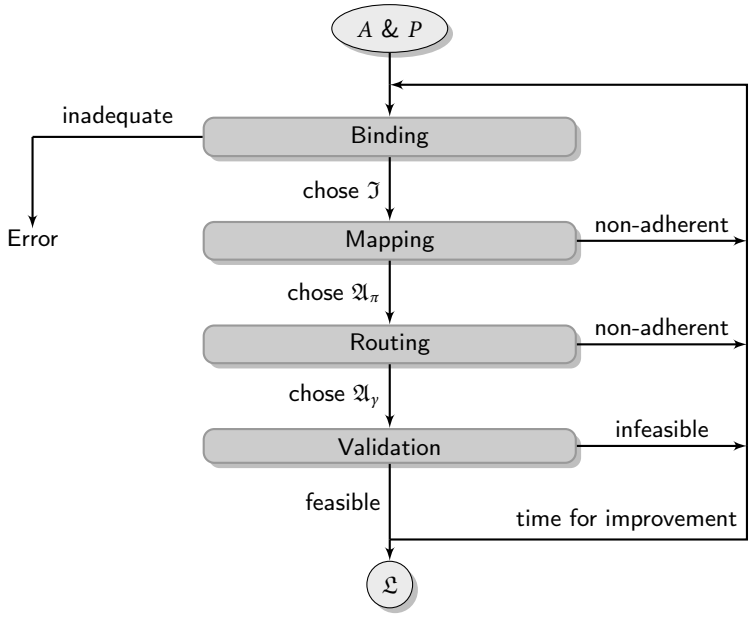


Figure 3.5 – Hierarchical search with iterative refinement

to light in later steps, we propose a strategy for iterative refinement. Figure 3.5 shows the hierarchical decomposition into steps used in our on-line spatial resource management algorithm for heterogeneous MPSoCs. We now describe each of these steps in more detail.

Binding

The goal of the first step is to choose an implementation (and thereby HwE type) for every task, i.e. to choose $\mathfrak{I}$ in $\mathfrak{L} = \langle \mathfrak{A}, \mathfrak{I} \rangle$. By choosing $\mathfrak{I}$ prior to $\mathfrak{A}_\pi$, this step implies a contract for $\mathfrak{A}_\pi$, i.e. adequacy limits the choice of $\mathfrak{A}_\pi(t)$ to HwEs of type τ, where $\mathfrak{I}(t) \in I_\tau$.

To prevent running into non-adherence directly after this step, we only consider those implementations for which an adhering mapping exists, i.e. that fit on at least one HwE in the platform. Thus, we only allow $\mathfrak{I}(t) = i$ when there is at least one HwE e of type τ, where $i \in I_\tau$ and all components of $\mathcal{C}(e) - \mathcal{R}(i)$ are ≥ 0.

The order in which we pick an implementation for each task is based on its *desirability*. We define the desirability of a task as the difference between the cheapest assignment and the second cheapest assignment of one of its implementations to an HwE. In other words, if the second best implementation is more expensive, the desirability to map the task increases.

To sustain the adherence of $\mathfrak{L}$, we virtually map the chosen implementation to the best-fitting HwE, which is determined in the desirability calculation. The implementation choice for the remaining tasks is affected by this mapping, as the available resource capacities in the platform are reduced. This guarantees that after this step (if this step manages to map all tasks), at least one $\mathfrak{A}_\pi$ exists such that $\mathfrak{L}$ is adherent, although $\mathfrak{L}$ might still be non-adherent due to the communication restrictions (i.e. no guarantees are given at this point with regards to the existence of an adherence preserving $\mathfrak{A}_\gamma$).

The algorithm for this step is based for a large part on Martello and Toth's bin packing heuristic [67, MTHG, pp. 206–207], with the exception that the desirability can be amended. We award penalties and bonuses to the desirability of some task(s) to prioritize their binding. Similarly, we can amend the cost of specific bindings. These amendments are the input of the feedback loop; two vectors (one for desirability and one for cost) with a component for every task in the application. We will return to the subject of feedback below.

Mapping

Resulting from step one, we have an implementation selector $\mathfrak{I}$ that assigns an implementation to every task in the application and we know that an assignment function $\mathfrak{A}_\pi$ exists, such that $\mathfrak{L} = \langle \mathfrak{I}, \mathfrak{A} \rangle$ is adequate. In this step, we take more detail into account, aiming at finding a task assignment $\mathfrak{A}_\pi$ with minimal cost. Besides cost factors based solely on the mapping of a task to an HwE, we also award assignments with a bonus for proximity of neighboured tasks in the application's task graph. This stimulates locality, causing the communication routes, assigned in the next step, to likely be short.

We define as starting point for the task assignment in the application's task graph, the task with the lowest communication degree. For this start task[3] t, we evaluate the costs of all possible assignments $\mathfrak{A}_\pi(t)$ to HwEs in the platform matching the type of the implementation the task was mapped to in the previous step. After assigning this task to the HwE with the lowest cost, we proceed with an iterative mapping process. At each iteration i, the tasks that lie in the task graph iso-distance i away from the start task are mapped. Using breadth first search starting from the HwEs to which the tasks from the previous iteration were mapped, we map each task to the best available HwE of the required type.

The search for HwEs can be stopped upon finding the first suitable set of HwEs (i.e. breadth-first, first-fit), but the search may be allowed to continue for a few more steps through the platform. How much further the search is allowed to continue is an input parameter of the algorithm. This is discussed in more detail in chapter 4.

[3] When multiple tasks have the same (lowest) communication degree, we can start with an initial *set* of tasks, or we can arbitrarily chose one from this set.

Again, we prevent immediate non-adherence in the next step, by only considering HwEs for a task that have sufficient communication resources to facilitate the task's communication requirements, at least, locally.

Routing

For the realization of step three, the channels are sorted by non-increasing throughput. Then, iteratively for each channel, a corresponding path is determined, taking into account the loads resulting from the previously mapped channels.

The sorting is done to increase the probability that a heavy demanding channel gets assigned a better path. In each iteration, for a given channel a shortest path between the source and destination interface of the channel is determined, where only those routers are taken into account which still have enough capacity for the throughput requirement of the current channel. Thus, an $\mathfrak{A}_\gamma$ is constructed iteratively, never overpacking communication capacities of a router.

In general, Uniform Cost Search (UCS)—esp. Dijkstra's algorithm—can be employed to find routes. For specific interconnection topologies, specialized algorithms can be employed. For example, in the case of regular mesh and torus NoCs, $A^\star$ does not suffer from its infamous unbounded memory behaviour, in the worst case it explores the same (amount of) nodes Dijkstra's algorithm does, and in the average case it outperforms Dijkstra's algorithm [81].

Adding $\mathfrak{A}_\gamma$ to the $\mathfrak{A}_\pi$ and $\mathfrak{I}$ from the first two steps, the result of this step is an *adherent* spatial mapping $\mathfrak{L} = \langle \mathfrak{A}, \mathfrak{I} \rangle$ where $\mathfrak{A} = \langle \mathfrak{A}_\pi, \mathfrak{A}_\gamma \rangle$.

Validation

The last step checks the throughput constraints posed on the application using techniques developed by Stuijk et al [33] and Hansson et al [44]. These techniques have been adapted to also allow latency constraints. For a detailed discussion of these adaptations, see section 4.4.

If it is detected that a constraint is violated, execution layout $\mathfrak{L}$ is *infeasible* and feedback should be given to higher steps to try to improve those characteristics of the mapping that violate the constraint(s). The constraint analysis performed in this step provides us with information with regards to throughput bottlenecks (by identifying *critical cycles* [53]).

If no constraint is violated, the constructed execution layout $\mathfrak{L}$ is *feasible*. In this case, possible points of improvement should be identified and fed back into the corresponding step.

Feedback

Tasks and channels on critical cycles detected during the validation process are prioritized, by increasing their desirability (for the binding step), or by fixing them to their fastest implementation. In general, a feedback immediately triggers a new iteration, to prevent that multiple changes influence the mapping process. In other words, if any step fails to find a satisfactory result, it immediately generates feedback so that 'higher' steps may generate a more suitable result. It is important to realize that this proposed iterative hierarchical approach differs significantly from simple local search methods and global-local search methods that are often used in heuristics. The feedback from a lower level may result in a completely different mapping on a higher level in a next iteration.

3.5 Conclusion

In this chapter, we gave formal definitions of the hardware components (HwEs, routers, links and interfaces), that together form a platform, and of the software components (tasks, implementations and channels), that together form an application. We have given definitions for resource capacities and resource requirements, that are each other's dual, such that they can be compared to see whether the resource requirements of a software component are met by the resource capacities of a hardware component. Furthermore, we have provided ways to composite, accumulate and threshold both capacities and requirements. Finally, we have defined QoS constraints on applications.

With these definitions, we have given definitions for what constitutes on-line resource management, i.e. finding an assignment of applications to platforms that meet the application's QoS constraints. Such assignments, called execution layouts, have been given qualitative metrics (adherence, adequacy and feasibility) and quantitative metrics (cost), so that they can be compared.

We have given an outline of a heuristic, that may be used to perform on-line resource management and shown how it relates to the qualitative metrics. The next chapters describe the heuristic and our implementation thereof in greater detail, relate it to the quantitative metrics and evaluate (by benchmark and case study) the heuristic.

KAIROS: AN OSRM IMPLEMENTATION

ABSTRACT – *The heuristic introduced in section 3.4 is implemented using algorithms described in more detail in this chapter. Algorithms for all levels of the hierarchical search are described, as well as the concrete implementation in a Linux kernel.*

4.1 BINDING

As discussed in chapter 3, the binding phase can be seen as an assignment problem. For every task, an appropriate implementation is selected. To avoid implementation selectors that make execution layouts inadherent (see chapter 3), besides the implementation selector, we construct a task mapping $\mathfrak{A}_\pi$ that does *not* take into account topological information from the platform (i.e. it can assign tasks to HwEs anywhere on the platform). This mapping is only constructed to guarantee that one exists. In the following steps this mapping is no longer used.

4.1.1 THE Bind ALGORITHM

The Bind algorithm (see algorithm 1) constructs implementation selector $\mathfrak{I}$. It is based on approximation algorithm MTHG, for the GAP by Martello and Toth [67, pp.

Parts and earlier revisions of this chapter have been published in [PhH:8].

Algorithm 1: Bind algorithm

```
   INPUT  : Task set T, implementation set I, HwE set E
   RESULT : An impl. selector ℑ for all tasks, such that a mapping 𝔄_π exists
1  η  ← ⟨⊥, ⊥, ⊥, ∞, ∞⟩                          ▷ field names: ⟨t, i, e, V, V′⟩;
2  T′ ← T;
3  𝔄_π ← ∅;
4  ℑ  ← ∅;
5  WHILE T′ ≠ ∅
6      B ← η;
7      FORALL t ∈ T′
8          C ← η;
9          FORALL {⟨i, e⟩ | i ∈ I(t), e ∈ E, Avail_⟨ℑ,𝔄_π⟩(e, i)}
10             V ← BindC(i, e);
11             IF V < C_V THEN C ← ⟨t, i, e, V, C_V⟩;
12             ELSE C_V′ ← MIN(C_V′, V);
13         IF C = η THEN FAIL ;
14         IF C_V′ − C_V > B_V′ − B_V THEN B ← C
15     𝔄_π ← 𝔄_π ∪ {⟨B_t, B_e⟩};
16     ℑ  ← ℑ  ∪ {⟨B_t, B_i⟩};
17     T′ ← T′ ∖ { B_t };
```

206–207]. The idea is that, one-by-one, an assignment is chosen for every task. Every possible assignment is associated with a cost. Since we are trying to find a minimal-cost assignment, the (locally) 'best' assignment for a task is that assignment with the lowest resulting cost. The order in which the tasks are assigned to implementations depends on their *desirability*. As in MTHG, we define the desirability of a task as the difference in cost between its best and its second-best assignment. Because every assignment of a task to an implementation can invalidate at most one assignment alternative for any other task[1], defining the desirability this way prioritizes task assignments based on highest increase in cost when alternatives are eliminated. The difference between MTHG and Bind lies in the choice of knapsacks, viz. a knapsack in Bind is a combination of an implementation with an HwE, as opposed to an implementation only, which is the case in MTHG.

In the bind algorithm (see algorithm 1), choices made are stored in records, referred to as binding choice records. The fields therein, as shown in the comment on line 1, are the task t being bound to implementation i on HwE e with cost V and the cost of the best alternative V'. The initialization (lines 1–4) defines the 'nil' value for a binding choice record η, the iteration task set T' and the resulting implementation selector $\mathfrak{I}$ and task mapping $\mathfrak{A}_\pi$. Binding choice records store the

[1] This assumes at most one implementation per HwE type, which is not a restriction per se, but in practice not much can be gained by making multiple implemenations for any HwE type.

task, implementation and HwE of a preferred assignment (resp. t, i and e), together with the cost associated with that assignment V and the cost of the second-best alternative V'. On every iteration of the main loop (lines 5–17), the best assignment B for the most desirable $t \in T'$ is added to the result (lines 15–16). When an assignment is chosen for a task, that task is not considered again (line 17).

The choice of the best assignment is made by considering all remaining unassigned tasks (lines 7–14). For each task, a candidate assignment C is sought, that assigns task $t \in T'$ to HwE $e \in E$ by selecting implementation $i \in I(t)$. C must still be *available*. Availability means the types of i and e are the same and e has sufficient resource capacity to meet i's requirements (Avail on line 9, discussed below). If there is any task for which no candidate assignments are found (line 13), Bind fails to find an implementation selector for an adherent execution layout. When the best candidate and the cost of the second-best candidate for a task are found, the desirability (the cost difference between the best and second best assignment) is computed and compared to that of the best solution found so far. If the candidate has a higher desirability, it replaces the former best alternative (line 14).

The inner loop (lines 9–12) first determines the cost of assigning implementation i to HwE e (BindC on line 10). Next, the last candidate is compared with that of the assignment under consideration and, if it improves the solution, replaces the candidate. When the current assignment does not improve the solution, its cost is still tested to see whether it improves the second-best alternative.

The cost function BindC must, of course, be 'simple'. We use simple profiling data that measures, per implementation, the energy consumption[2] for specific types of HwEs, i.e. a straightforward lookup. However, when managing HwEs that have large capacities (with regards to the requirements of tasks) and where multi-tasking is very common, a typical added component in this cost function could take into account the remaining capacities of the HwE, so as to achieve cross-HwE-type load balancing. In the complexity analysis of Bind (see section 4.1.2), we assume constant complexity for the cost function.

The availability (Avail) of a binding of task t with implementation $i \in I(t)$ depends on the existence of an HwE $e \in E$ that is of the correct type, i.e. $\tau(t) = \tau(e)$, and with sufficient capacities, i.e. $\mathcal{C}_e - \mathcal{R}_i \geq 0$. Since the implementation selector is constructed incrementally in Bind, earlier bindings were tested already for availability. Newly selected bindings may not make these older bindings unavailable. Therefore, when an HwE is used to show that a binding is available, the task in the binding is also mapped to this HwE (line 15). The mapping of all previously bound tasks is assumed to be fixed when determining the availability of new bindings. This is indicated in the algorithm by subscripting the implementation selector and task assignment function (line 9).

[2] The average energy consumption when measured using representative input data.

Because every iteration of the outer loop reduces T' by one element, the loop within it (lines 7–14) decreases in length. More precisely, the outer two loops combined have $\frac{1}{2}|T|(|T|+1)$ iterations. The set of binding candidates (line 9) has a worst case size of $|I \times E|$. When the complexity of Avail is constant, the worst case time to compute this set is proportional to its size. Thus, the Bind algorithm as discussed has a worst case time complexity of $\mathcal{O}(T^2IE)$.

This complexity can be reduced by observing that every binding choice invalidates at most one binding candidate for any other (unbound) task. When a binding candidate is invalidated, the corresponding task may have lost its best alternative (in which case this should be recalculated), its second best alternative (in which case the task's desirability changes), or an alternative that changes neither the best alternative, nor the desirability. This means the average case time complexity of Bind can be greatly improved by having an ordered representation of T' that is only rearranged for tasks that lose their best or second best candidate. Also, since the HwEs in a platform are static, the set of implementation-HwE combinations can be preprocessed, e.g. when an application is installed on a platform. This eliminates type mismatched combinations. Such a preprocessed set of assignable implementations, denoted by $I^\star$, contains members with constrained capacities, but no type constraints. If an added memory size of $\mathcal{O}(TI^\star)$ is acceptable, for every task, the order of all candidates can be stored after being calculated once. This calculation takes $\mathcal{O}(TI^\star \log I^\star)$ time [67, pp. 207–208]. The order of alternatives per task is never influenced by Bind, so maintaining order becomes very simple. This reduces the worst case time complexity of Bind to $\mathcal{O}(T^2 + TI^\star \log I^\star)$.

4.2 Mapping

Given an implementation selector $\mathfrak{I}$, the assignment of tasks to HwEs is still a complex problem. Most single-chip production platforms to date are relatively small. With increasing integration and new, more scalable architectural building blocks (NoC, etc.), inter-HwE distance begins to get an essential influence in mappings of tasks to HwEs. Having to take into account the distance between the HwEs to which two communicating tasks are assigned only raises the complexity.

The heuristic we propose for the mapping phase takes these observations into account. First, it partitions the set of tasks into classes. Next, it iterates over these classes, mapping every task in a class to an HwE. Tasks are in the same class if they are in the same order neighbourhood of a given starting point. By iterating over the classes in increasing distance from the starting point, and by mapping tasks from a class close to those from the previous class, locality is promoted. The remainder of this section describes this heuristic in more detail.

We apply a divide-and-conquer [63] approach to partition the mapping problem into variable-sized sub-problems. The idea is as follows: An application $A = \langle T, C \rangle$ is iteratively partitioned into subsets. On iteration i, all tasks in subset $T_i \subseteq T$ are mapped onto a subset of the HwEs $E_i \subseteq E$. Next, the subset T_{i+1} is constructed by finding all unmapped tasks that communicate with tasks in the current iteration's subset T_i. The following describes this process in more detail, after introducing a few concepts and their notation.

Degree, neighbourhood and cumulative sets

The *degree* of a vertex v, denoted by $d(v)$, in graph G (viz. task t in application A, or HwE e in pathed platform $P^\star$) is defined as the number of edges incident to v in G (regardless of their direction). When direction *is* relevant, we denote the *out-degree*—the number of edges from v—as $d^+(v)$ and the *in-degree*—the number of channels to v—as $d^-(v)$, i.e. $d(v) = d^+(v) + d^-(v)$. Furthermore, we define the *maximum degree* $\Delta(G)$ and the *minimum degree* $\delta(G)$ as the largest, resp. smallest degree of vertices in G.

The *neighbourhood* of a vertex v in graph $G = \langle V, E \rangle$, denoted by $N(v)$, is the set of vertices directly connected to v in graph G, disregarding edge direction, viz.

$$N(v) = \{v' \in V \mid v \neq v', \langle v, v' \rangle \in E \vee \langle v', v \rangle \in E\}$$

This is also referred to as the *first-order* neighbourhood. The i^{th}*-order* neighbourhood of v in the underlying undirected graph $\tilde{G}$ of G, denoted by $N_i(v)$, is the set of vertices in $\tilde{G}$ for which the *shortest* path from v has length i, where the length of a path is the number of edges on it. Analogously to degrees, neighbourhoods can be expressed with direction as $N_i^+(v)$ for the *out-neighbourhood*—those vertices to which the shortest path from v has length i—and as $N_i^-(v)$ for the *in-neighbourhood*.

For the sake of a simple notation, we introduce a shorthand for *cumulative sets*, to indicate the union of all sets with smaller-or-equal index, i.e.

$$N_i^\star(v) = \bigcup_{j=0}^{i} N_j(v)$$

Here $N_0^\star(v) = \{v\}$. In the case of neighbourhoods, this indicates all vertices that have a shortest path of length *at most* i. However, we use the $\star$ as a general notation for cumulative sets, so for the classes of the partitioned task set, $T_i^\star$ indicates all tasks mapped in iterations up-to-and-including i. As a further notational simplification, we say that when a function that is defined on elements of a set is applied to a set, it is applied to all elements, viz.

$$N(V) = \bigcup \{N(v) | v \in V\}$$

Starting point

In many cases, especially in embedded systems, the mapping of some tasks of the application is fixed. Most commonly, this is because input and output have to come from predetermined HwEs of the platform (because that HwE is an input or output of the platform from or to the outside world, e.g. an A/D-converter). Formally, we say that the mapping of a task t is fixed after the binding phase, if there is precisely one HwE e on the platform capable of executing the implementation selected for t, i.e. given an implementation selector $\mathfrak{I}$, the mapping $\mathfrak{A}(t)$ is considered fixed when $\exists!\, e \in E\,[\tau(e) = \tau(\mathfrak{I}(t))]$. Because capacities are not taken into account for this criterion, a fixed mapping imposes only a (constant time) lookup for that task.

For the mapping algorithm, the initial subset of tasks T_0 is defined as the set of all tasks that have a fixed mapping after the binding phase. For these tasks, the corresponding initial set of HwEs E_0 is already specified, so no further initialization is required. If no fixed tasks exist, we choose T_0 to be a singleton set consisting of a randomly chosen task from the subset of tasks with the lowest degree, i.e. any task $t \in T$ such that $d(t) = \delta(T)$. Having a low degree in the task graph makes a task a 'safe' starting point, because if it is mapped onto an HwE surrounded by unavailable HwEs, the chances that all required links may still be routed are higher than for those of a task with a high degree (because there are less links to route). For the chosen task t to form the initial class, i.e. $T_0 = \{t\}$, an HwE has to be found to form the singleton set $E_0 = \{e\}$, such that t is assigned to e. We find it by minimizing the differences between the capacities in the neighbourhood of e and the requirements in the neighbourhood of t in order to find the tightest fit, i.e. we chose e by

$$\min_{e \in E} \left| \sum_{e' \in N(e)} \mathcal{C}_E(e') - \sum_{t' \in N(t)} \mathcal{R}_I(\mathfrak{I}(t')) \right|$$

By initializing the algorithms in this way, the external fragmentation of HwEs is reduced, which increases the probability that an allocation for applications that are started after the current application exists. External fragmentation is defined formally as the fraction of pairs of adjacent HwEs, for which one is available for a task and the other is not. Since the aim of the minimisation of external fragmentation is to contribute to the probability that *other* applications can be started on the same platform, the definition of external fragmentation f_{ext} should not depend on the task set of the application currently under consideration. Therefore, we use the minimum capacity vectors (see section 3.2.3) as a threshold for availability, viz.

$$\begin{aligned} \alpha &= \left| \left\{ \langle e, e' \rangle \mid e \in E, e' \in N(e), \mathcal{C}_E(e) \geq \mathcal{C}^{\min}_{\tau(e)}, \mathcal{C}_E(e') < \mathcal{C}^{\min}_{\tau(e')} \right\} \right| \\ \theta &= \frac{1}{2} \sum_{e \in E} |N(e)| \\ f_{ext} &= \frac{\alpha}{\theta} \end{aligned}$$

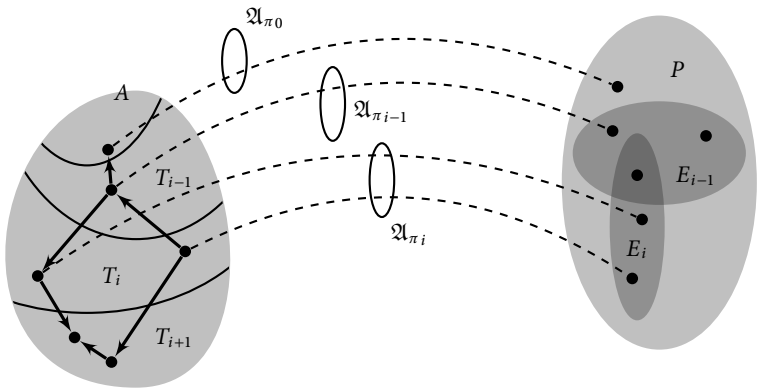


Figure 4.1 – Incremental mapping approach

This concrete measure of external fragmentation is not only used in the initialization, but also in the cost function for every assignment, as demonstrated below.

Consecutive iterations

Having initialized the algorithm with an initial assignment $\mathfrak{A}_{\pi 0} : T_0 \to E_0$, consecutive iterations must find assignments for the partitions of the task set, until the entire task set is mapped onto HwEs. The tasks are partitioned iteratively, i.e. starting with the above constructed T_0 for any T_{i-1}, we construct the next partition T_i. To promote locality of communication, the unmapped tasks that communicate with a mapped task should be mapped as close as possible to the HwEs the tasks of T_{i-1} are mapped to. This is why, given a mapping for T_{i-1}, the next partition T_i is chosen as the set of tasks with which the tasks in T_{i-1} communicate. Intuitively, this suggests that $T_i = N(T_{i-1})$, albeit that this definition includes tasks on a cycle in the task graph, or tasks reachable by paths of different lengths from other tasks, in multiple classes. Instead, the partitions can directly be defined in terms of the initial partition, as $T_i = N_i(T_0)$. This approach is illustrated in figure 4.1.

Since the complexity of finding a mapping for a single class T_i still grows rapidly in the size of the HwE set, E is also partitioned into classes. The class E_i is created by searching for HwEs available for tasks in T_i, until T_i is covered, i.e. for every task $t \in T_i$ there is an available HwE $e \in E_i$. To stop searching immediately after finding a covering set of HwEs only allows for cost minimization. Other objectives, like minimazation of external fragmentation, may require more room for choice in the mapping from T_i to E_i. To this end, E_i is expanded a little after a covering set is found for T_i.

The search for a covering set of HwEs is also realized as an iterative process, in

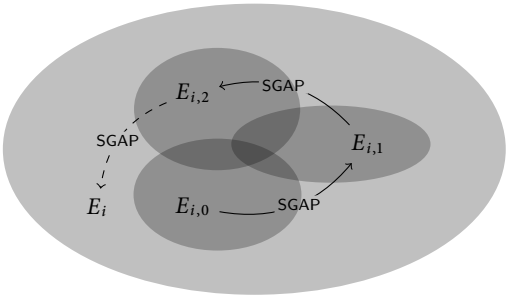


Figure 4.2 – Incremental search for HwEs

which, as before, locality is increased by taking into account the HwEs close to HwEs to which the tasks in T_{i-1} are mapped. In other words, the search starts from ran $(\mathfrak{A}_{\pi\, i-1})$, where ran$(f)$ denotes the range of function f. During iteration j, HwEs in the j^{th}-order neighbourhood of this starting point, denoted by $E_{i,j}$ are considered for addition into E_i. The direction of the communication between tasks in T_{i-1} and tasks in T_i is taken into account by using the directed neighbourhood. Searching terminates a predefined number of iterations after a partition E_i has been found that covers T_i. In other words, if the set $(E_i)^\star_n$ constructed in iteration n covers T_i and the predefined number of added iterations is k, then $E_i = (E_i)^\star_{n+k}$.

To determine whether an HwE set $(E_i)^\star_j$ covers task set T_i, an assignment is sought from tasks to HwEs. This is a reduced version of the original GAP to assign all tasks in T to E. It is solved by an approximation algorithm SGAP, discussed in more detail below. When SGAP fails to find a mapping from all tasks in T_i to HwEs in $(E_i)^\star_j$, the latter is assumed to not cover the former and the search continues and constructs $E_{i,j+1}$. On the other hand, when SGAP succeeds, searching continues for the aforementioned predefined number of iterations. Applying SGAP to this set results in $\mathfrak{A}_{\pi\, i} : T_i \to E_i$. This process is illustrated in figure 4.2.

Algorithm

The Map algorithm (see algorithm 2) constructs a task assignment $\mathfrak{A}_\pi$. Above, the initialisation is described in terms of initial task and HwE sets. In lines 1–2, however, the initial task assignment $(\mathfrak{A}_{\pi 0})$ is constructed as a whole. Its domain is the initial task set and its range is the initial HwE set, i.e.

$$\begin{aligned} \text{dom}\,(\mathfrak{A}_{\pi 0}) &= T_0 \\ \text{ran}\,(\mathfrak{A}_{\pi 0}) &= E_0 \end{aligned}$$

The outer loop of the algorithm (lines 3–10) iteratively maps the partitions (line 4) of the task set, until all tasks have been mapped. The partition of the HwE set

Algorithm 2: Map algorithm

```
   INPUT  : An application A = ⟨T, C⟩, impl. selector ℑ, platform P = ⟨E, L⟩
   RESULT : A task assignment 𝔄_π for all tasks in T
1  𝔄_π0 ← {⟨t, e⟩ ∈ T × E | {e} = {e′ | e′ ∈ E, τ(e′) = τ(ℑ(t))}};
2  IF 𝔄_π0 = ∅ THEN 𝔄_π0 ← {⟨t, e⟩ ∈ T × E | d(t) = δ(T), Avail(e, ℑ(t))};
3  WITH i ∈ ℕ LOOP
4  |  T_i ← N_i(dom(𝔄_π0));
5  |  WITH j ∈ ℕ LOOP
6  |  |  E_i,j ← ⋃_φ N_j^φ(ran(𝔄_π i−1 ↾ N^φ(T_i)));
7  |  |  IF E_i,j = ∅ THEN FAIL ;
8  |  |  𝔄_π i,j ← SGAP(T_i, (E_i)*_j);
9  |  UNTIL T_i = dom(𝔄_π i,j−k) THEN 𝔄_π i ← 𝔄_π i,j
10 UNTIL T*_i = T THEN RETURN 𝔄*_π i
```

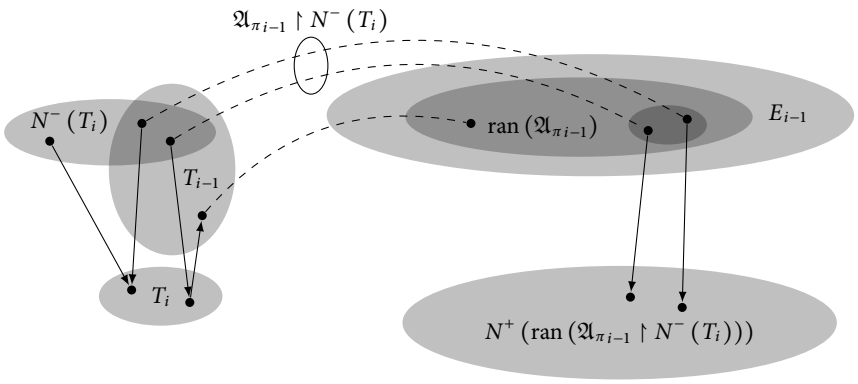


Figure 4.3 – Search iteration for $E_{i,j}$

E_i is constructed by iterative searching (lines 5–9) until k iterations after the first covering set was found (hence $\mathfrak{A}_{\pi i,j-k}$ on line 9). The search starts with the HwEs to which the tasks from the previous partition are mapped ($\text{ran}(\mathfrak{A}_{\pi i-1})$), taking into account the direction of inter-task communication.

An example is illustrated in figure 4.3. Tasks in T_i receive communication from tasks in $N^-(T_i)$, some of which are—in this example—part of T_{i-1}, i.e. have been mapped in the previous iteration. When restricting the task assignment function $\mathfrak{A}_{\pi i-1}$ to $N^-(T_i)$ and taking its range, we find the HwEs to which tasks are mapped that send communication to the tasks we want to currently map. The out-neighbourhood of these HwEs are good candidates to map tasks from T_i to. Similarly, for HwEs to which tasks from $N^+(T_i)$ are mapped, the in-neighbourhood are good candidates

for those tasks in T_i that send communication to tasks in T_{i-1}. Since the first-order neighbourhood may not contain sufficient HwEs to map T_i to, $E_{i,j}$ is the union of the j^{th}-order directed neighbourhoods, as described above. In the algorithm, the union of both directions has been abbreviated to ϕ, to mean either + or −, and to its compliment $\overline{\phi}$, to mean either − or +, respectively.

When no further candidate HwEs are found, the algorithm fails (line 7). A concession has been made here to notational brevity: In actual implementations, if a covering set was found in a previous iteration (i.e. the current iteration is one of the k extra iterations), the mapping constructed for that covering set is assigned to $\mathfrak{A}_{\pi i}$. A mapping is sought on the cumulative set $(E_i)^\star_j$, since the subsets $E_{i,j}$ are the steps by which E_i is iteratively extended.

Algorithm 3: SGAP algorithm

INPUT : Partition of the task set T_i and candidate partition of the HwE set $(E_i)^\star_j$
RESULT : A (possibly partial) mapping $\mathfrak{A}_{\pi i,j} : T_i \to (E_i)^\star_j$

1 $\mathfrak{A}_{\pi i,j} \leftarrow \varnothing$;
2 $c^1 \leftarrow \overrightarrow{\infty}$;
3 FOR $e \in (E_i)^\star_j$
4 FOR $t \in T_i$ $\left(c^2(t) \leftarrow \mathsf{MapC}(t, e)\right)$;
5 REPEAT
6 $B \leftarrow \langle \perp, 0 \rangle$ ▷ field names: $\langle t, V \rangle$;
7 FOR $C \in \left\{ \langle t', c^1(t') - c^2(t') \rangle \mid t' \in T_i, \mathsf{Avail}(e, \mathfrak{I}(t')) \right\}$
8 IF $C_V > B_V$ THEN $B \leftarrow C$
9 IF $B_t \neq \perp$ THEN
10 $c^1(B_t) \leftarrow c^2(t)$;
11 $\mathfrak{A}_{\pi i,j}(B_t) \leftarrow e$;
12 UNTIL $B_t = \perp$;

The SGAP algorithm (see algorithm 3) is an adaptation from [20]. For every HwE in $(E_i)^\star_j$, it tries to find an optimal packing of tasks from T_i. When a task occurs in the optimal packing of two HwEs, it is moved to the HwE that imposes the lowest cost. This is accomplished by keeping track of the best-case cost for every individual task (c^1) and per combination of HwE and task, the cost of mapping the task to the HwE (c^2).

The outer loop of the algorithm (lines 3–12) traverses the HwEs in $(E_i)^\star_j$. First, the cost of mapping any task from T_i to the current HwE is calculated and stored in c^2 (line 4). Next, the inner loop (lines 5–12) iteratively choses the best task to map onto this HwE (lines 7–8) and if a task is found, i.e. if a task exists that still fits this HwE, the cost of this mapping is stored as the best for the task and the mapping is recorded (lines 9–11). The cost for this mapping is indeed an improvement on any

mapping of this task found earlier, because the measure of profit by which the best task to map is the improvement of the best-case cost found until now ($c^1(t') - c^2(t')$ on line 7).

As explained above and illustrated in figure 4.2, every set $(E_i)^\star_j$ is a strict superset of $(E_i)^\star_{j-1}$. Thus, when SGAP fails to find a complete mapping for the tasks in T_i during iteration $j-1$, it is invoked again with a larger set of HwEs. Since the task set T_i is unchanged between these iterations, instead of starting with an empty initialization (as is the case in lines 1–2), SGAP can be initialised with the results from the previous invocation.

Algorithm 4: MapC algorithm

```
   INPUT  :
   RESULT :
1  IF Avail (e, 𝔗(t)) THEN
2  |  m_1 ← |N(e) ∩ 𝔄*_{π_i}(N(t))|;
3  |  m_2 ← |N(e) ∩ 𝔄*_{π_i}(T)|;
4  |  m_3 ← |{e' ∈ N(e) | Active(e')}|;
5  |  f ← 3|N(e)| − Σ_{i=1}^{3} m_i;
6  |  c ← Σ{D(e, e') | e' ∈ 𝔄*_{π_i}(N(t))} + U|N(t) ∖ dom(𝔄*_{π_i})|;
7  |  RETURN f W_frag + c W_comm;
8  ELSE RETURN ∞;
```

Finally, the MapC() algorithm (see algorithm 4) expresses the cost model used. When HwE e is unavailable for the execution of a task, the cost of assigning that task to the HwE is infinite (line 8). Otherwise, if the HwE *is* available for the task, the cost of the assignment consists of two components (line 7): one for fragmentation and one for communication (where W_{frag} and W_{comm} are the weights assigned to them, respectively, as part of the configuration of the resource manager).

The penalty for fragmentation f is given to those mappings that increase fragmentation. This means that this penalty is highest when a task is mapped to an HwE whose neighbours are all inactive. The penalty costs are decreased (by one) for each neighbouring HwE e for which one of the following three conditions holds: When a neighbour task of t is assigned to a neighbour HwE of e (m_1), when any task from the same application is mapped to a neighbour HwE of e (m_2) and when a neighbour HwE of e is already assigned any task from any application (m_3), i.e. when it is active. These criteria are overlapped, i.e. when a task communicating with t is mapped to $e' \in N(e)$, all three criteria hold. Because the best case should not impose a penalty at all, the worst-case penalty is set to $3|N(e)|$ and the best-case may be 0.

The communication component is the sum of distances between the HwEs to which the tasks in the neighbourhood of t are mapped and HwE e. However, since E_i is iteratively explored, not all inter-HwE distances may have been found yet: $E_{i,j}$ is an

extention from $E_{i,j-1}$ with HwEs from $N(E_{i,j-1})$, it may contain an HwE $\dot{e}$ that is in $N_j(e)$ and in $N_n(e')$, where $n > j$, for some $e, e' \in E_0$. This means that on iteration j, a route from e to (or from, depending on the direction of the neighbourhoods) $\dot{e}$ has been found already, but a route from e' to $\dot{e}$ has not. Searching for that (yet) unknown route requires running a routing algorithm. To avoid the cost of this added searching, we only look at distances of routes between HwEs that have already been explored. During exploration, distances found are stored in a distance matrix D. Any distance not previously found is set to a large constant U to indicate the penalty for uncertainty. This means that the value of $D(e, e')$ on line 6 should be read as

$$D(e, e') = \min_{k \leq j} \left(\{ k \mid e \in N_k(e') \} \cup \{ U \} \right)$$

The second term of the communication component states that for neighbouring tasks that have not yet been mapped, we also give penalty U.

4.2.2 Complexity of Map

In the worst-case scenario, all HwEs of a platform are considered. This happens, for example, when a task $t \in T_{i-1}$ is mapped to an HwE $e \in E_{i-1}$ and the first HwE e' of the required type for $t' \in T_i$, i.e. $\tau(e') = \tau(\mathfrak{I}(t'))$, is in $N_j(e)$ where $(E_i)^\star_j = E$. Thus, the complexity of Map includes a term $|E|$.

Map iterates over the classes T_i of the partitioning of T (algorithm 2, lines 3–10). In every iteration of SGAP, the best task to assign to the current HwE is selected (algorithm 3, lines 7–8). By sorting all tasks in T_i by their cost improvement ($c^1(t) - c^2(t)$ on line 7), the best task can be found in $\mathcal{O}(\log T_i)$.

This leads to a total complexity of $\mathcal{O}(E \sum_i T_i \log T_i)$ in which the term T_i can not be bound any tighter than T. Thus, we derive for the execution time complexity of Map:

$$\mathcal{O}\left(E \sum_i \left(T_i \log T_i \right) \right) \leq \mathcal{O}\left(ET \log T \right)$$

4.3 Routing

The realisation of the routing part is based on known routing algorithms. Therefore, this section does not give a detailed explanation of algorithms. Instead, we give a concise overview of the relevant considerations and choices available for implementations, as well as the choices made for the implementation described in section 4.5.

4.3.1 Considerations

Communication networks generally identify three classes of communication, i.e. broadcast (one-to-all), multicast (one-to-many) and unicast (one-to-one). Broadcast is often used in systems where the operating system can perform ad-hoc management of one or all running processes. In real-time systems, however, this is usually not employed, because it introduces hard to predict timing issues [16, pp. 15–18]. Multicast communication is useful for applications in which the same data is sent to a number of different tasks, especially when that data is voluminous. On-chip router architectures that support multicast routing are rare, because the cost in terms of area is considerable. However, for applications such as the beamformer described in chapter 5, they can considerably reduce redundant traffic and, thereby, energy consumed in communication. As an alternative to multicast routers, busses can facilitate multicast communication, although they offer considerably less scalability and make a platform considerably more application specific [52].

Unicast communication is the most common in real-time streaming applications. Each channel in an application is considered a point-to-point communication channel, that requires a unicast route through the interconnect. When two communicating tasks from an application are mapped to different HwEs, a route needs to be found from one HwE to the other. The cost function used to express the cost of a route can be chosen to reward, for example, low energy consumption or low network congestion [54]. Since the mapping of tasks already stimulates locality, we use the path length (in terms of routers and links) as a measure for cost.

The optimal routing of all communication channels of an application requires the simultanious consideration of all channels. The complexity of such a holistic solution to the routing problem is prohibitively complex. We chose to route channels one at a time (similar to [68]). Depending on what interconnection resources (e.g. bandwidth) are most scarce, channels can be sorted in terms of their requirement of the most scarce resources, descending. On platforms where the number of connections is most scarce, as those in chapter 5, however, sorting only imposes additional cost and renders no improvement of the final routing.

4.3.2 Routing algorithms

The routing algorithm employed in the implementation of the on-line spatial resource manager (see section 4.5) is UCS. UCS is a modified version of Dijkstra's shortest path algorithm [29], which terminates as soon as the destination is reached. This algorithm does not allow for non-positive cost links, thus there is a restriction on any given bonusses for a link: A bonus (leading to a reduction of the cost) must be strictly smaller than the cost[3].

[3] Should such bonusses be deemed necessary, the Bellman-Ford algorithm [9] may be used.

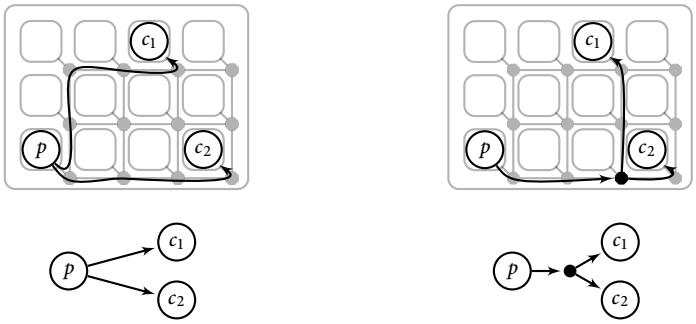


Figure 4.4 – Shorter *cummulative* communications paths with multicast routing

Depending on the architecture of the platform's interconnect, more specialized routing algorithms can be used. As an example, consider that for highly regular interconnect topologies, such as meshes and tori, $A^\star$ *search* [25] has the same worst-case complexity as UCS, but a far better average-case performance. The most commonly known weakness of $A^\star$, the exponential memory complexity, is not a problem for topologies with a relatively low number of nodes (dozens to a few hundred).

4.3.3 Multicast routing by rendezvous points

As mentioned above, when one task produces identical input for multiple other tasks, both interconnect resource requirements and energy consumption can be reduced by having multicast routers. If such routers exist in the interconnect, the on-line spatial resource manager needs some extra facilities to deal with this.

To this end, task graphs are extended to have 'router tasks' that have one incoming and $n > 1$ outgoing channels and can be mapped to a multicast router. In the mapping phase, these tasks are ignored, i.e. a router task is transparently perceived as n independent channels. After the mapping phase, the same algorithm (Map) is used to map the router tasks to multicast routers. If the resulting cummulative path length is shortened by using a multicast router (figure 4.4), the router task is accepted. When this is not the case, the router task is removed from the task graph and rewritten to n channels.

4.4 Validation

As with routing, the realisation of the QoS validation part is based on known routing algorithms. Therefore, this section does not give a detailed explanation of algorithms. Instead, we discuss how task graphs are rewritten to SDF graphs, using

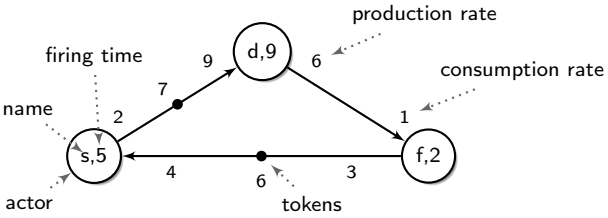


Figure 4.5 – Example sdf graph

information from the binding, mapping and routing stages, and which analysis is performed on these sdf graphs to decide whether or not an execution layout is feasible. We will, however, first give a brief introduction to sdf graphs and the representations used.

4.4.1 synchronous data flow graphs

sdf graphs are directed graphs, consisting of vertices, referred to as 'actors', edges and a few annotation functions. Actors are annotated with their *name* and their *firing time*. Edges represent communication and arbitration dependencies between actors. Edges are annotated with three values: The number of *tokens* they contain, the *production rate* and the *consumption rate*. The production and consumption rates are static parameters, i.e. they do not change over time. The number of tokens contained on an edge can change, as described below.

A token is the atomic unit of data in sdf. Tokens are produced onto an edge at the *end* of the firing of the producing actor (at the edge's start). Tokens are consumed from an edge at the *beginning* of the firing of the consuming actor (at the edge's end). How many tokens are produced and consumed *per firing* is indicated by the production and consumption rates with which the edges are annotated.

An actor may fire when on every incoming edge there are at least as many tokens as the edge's consumption rate annotation. When an actor fires, it consumes the number of tokens given by the consumption rate on its incoming edges, and after the actor's firing time, it produces the number of tokens given by the production rate on its outgoing edges. Consumption and production of tokens occurs in zero-time. Consecutive, firings of an actor may overlap, i.e. a new firing may start as soon as all incoming edges contain sufficient tokens, regardless of when previous firings started. A common technique to model tasks that do not run multiple parallel instances is to add a self-edge, i.e. an edge that begins and ends at the same actor, with one initial token on it, a production rate of one and a consumption rate of one. For tasks that allow n parallel instances of itself, a self-edge with production and consumption rates of one is added with n initial tokens.

We use a graphical notation for sdf graphs, such as the example in figure 4.5. In this section, we draw sdf graphs only at time moments, in which no actors are firing.

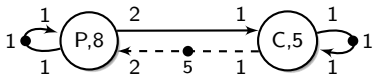


Figure 4.6 – Example model of FIFO communication

This implies that all available tokens are expressed in the figures.

For every SDF graph, a *repetition vector* q can be determined. A repetition vector denotes the smallest number of firings for each actor, such that the production and consumption of tokens for all edges are balanced. In other words, for an edge from actor s to actor d with production rate p and consumption rate c, it holds that $p \cdot q_s = c \cdot q_d$.

Modelling

We use a few modelling techniques to express (elements of) execution layouts as SDF graphs. As is common practice in embedded system applications, a task is expressed as a single actor[4]. An actor in an SDF graph may fire infinitely many times simultaneously, but tasks are executed on finite resources and thus require a further constraint in the model. The actors modelling tasks therefore have a 'self-cycle', i.e. if actor a is modelling a task, the SDF graph contains an edge $\langle a, a \rangle$ with production and consumption rate 1.

First-In First-Out (FIFO) buffers can be modelled using two edges, where one models the data flowing in one direction and the other models free buffer space flowing in the opposite direction. An example of an SDF graph modelling two actors, one producer (P) and one consumer (C), communicating through a FIFO buffer with a capacity of 5 tokens is shown in figure 4.6. As a visual reminder of the fact that we model a buffer, the edges that model free space are dashed. In terms of SDF graphs however, they are normal edges.

There are different programming models (e.g. [12]) that can be translated directly to this modelling technique. However, it is quite common practice to manually derive an SDF model. How the SDF graph is obtained is beyond the scope of this thesis.

4.4.2 Rewriting task graphs

Because tasks are modelled with a single actor, the task graph already provides the initial structure of the SDF graph. Per stage of the heuristic search method, the

[4] As opposed to, e.g. a common application of SDF graphs in compilers, where actors usually correspond to blocks of code.

choices made in that stage are translated into changes of the final SDF graph. In the current implementation (see section 4.5), the production and consumption rates are fixed for every channel. For increased precision and flexibility, the communication granularity should be allowed to vary from one implementation to the next. The rewriting operations per stage are as follows:

1. Binding: Having selected implemenations for all tasks, tasks' actors can now be annotated with their implementation's execution time as their firing time. The execution time of an implementation is the time it would cost to execute that implementation given exclusive access to a HwE of the corresponding type. The number of tokens on an actor's self-edge is determined by an implementation's internal parallelism (i.e. how many consecutive executions can be started before finishing their current execution). In most cases, only one execution is possible simultaneously.
2. Mapping: With the mapping of tasks (and thereby implemenations) to HwEs, every implementation can now be scheduled on its HwE. The execution time from the previous step, with the scheduler settings from the HwE a task is mapped to, can be translated to a *response time* [104], which compensates for arbitration overhead. Actors' firing time annotation is changed to this response time.
3. Routing: When channels are mapped onto paths, the temporal behaviour of the interconnection components on a path need to be modelled in SDF. Quite a lot of work has been done for different architectures of interconnect [42, 43, 52] and there are general models [104] for resources that fall under a specific class (e.g. latency-rate servers, budget servers). Using these models, for every channel in the task graph, there is a pair of edges in the SDF graph that needs to be rewritten to express the effects of routing.

The rewriting for the routing stage needs to take into account effects (esp. on latency) caused by communicating through the interconnect. Many interconnect architectures contain FIFO buffers [22, 40, 42, 52], but the total buffer capacity on a path through the interconnect is typically at least an order of magnitude smaller than buffers intrinsic to the application [12]. When buffers are too small, applications may deadlock. Increasing the size of buffers in a deadlock-free application never introduces deadlock, weakly increases throughput and increases maximum latency [42, 101, 102].

Given these considerations and the fact that the routing stage constructs routes that provide sufficient throughput for the routed channel, we do not need to explicitly model the buffer capacities and throughput limitations of routed channels. We do, however, need to model the latency, especially because this can form a throughput bottleneck when the task graph contains cycles (directed or undirected). Obviously, data having to traverse a route imposes latency, but the reverse flow of free space may also experience latency [52], e.g. when the route's buffers are distributed over multiple routers. An example of the rewritten SDF graph of the example in figure 4.6 to include routed communication is depicted in figure 4.7. Actors P and C occur as

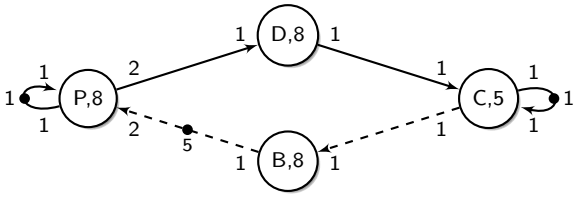


Figure 4.7 – Example model of two mapped tasks with a routed channel.

they were. The edges between them have been replaced by a model of the *routed* channels. Actor D models the latency imposed on messages from P to C. Actor B models the latency imposed on the notification of available buffer space from C to P.

4.4.3 Throughput analysis

To determine the guaranteed throughput of an application, we perform a simulation of the behaviour described by the SDF graph that is obtained from rewriting the task graph. To this end, we assume that the SDF graph executes in a *self-timed* manner, i.e. that every actor fires as soon as it is enabled [44]. In other words, we assume the SDF graph describes the complete temporal behaviour of the application.

The simulation consists of the traversal of *graph states*. A graph state consists of the number of tokens on each edge and the list of firing actors with their time until completion. For every time unit, first, all actors to complete a firing produce their tokens and, second, all actors that are now enabled start firing. The simulation continues until a graph state is reached that has been reached before, i.e. a cycle is found.

Many streaming applications have some sort of start-up behaviour before they show the expected (quasi-)periodic behaviour. In terms of our simulation, this means that there are a few graph states *not* part of the cycle. These 'start-up' states are referred to as the *transient phase*, whereas the cycle of graph states to follow is referred to as the *periodic phase*. From any state in the periodic phase, letting every actor fire as many times as indicated by the repetition vector returns the graph to the same state. Passing through all states of the periodic phase once is called a *graph iteration*. Because we assume the periodic phase to last orders of magnitude longer than the transient phase, i.e. the transient phase is considered incidental, QoS guarantees are given for the periodic phase only.

Now we can determine the throughput of the SDF graph. The throughput of any actor of the SDF graph is equal to the average number of firings per time unit in the periodic phase of the self-timed execution. Thus, the throughput of an actor is determined by counting the number of times it fires in the periodic phase, divided

by the duration of one traversal of the cycle of states. The throughput of the entire SDF graph is defined as the throughput of any of its actors, divided by that actor's number in the repetition vector. For detailed discussions and proofs of this method, we refer to [33].

4.4.4 LATENCY ANALYSIS

Latency constraints are typically constraints on endpoints of an application, i.e. constraints on the latency between some input and some output. Thus, modelling latency involves a relation between two actors; source and sink. There are two ways to look at latency constraints: the Early Latency Response (ELR) and the Late Latency Response (LLR). When all actors in the graph fire as many times as indicated in the repetition vector, ELR is the time between the start of the first firing of the source actor and the *start* of the *first* firing of the sink actor, whereas LLR is the time between the start of the first firing of the source actor and the *completion* of the *last* (as indicated by the repetition vector) firing of the sink actor.

The SDF graph can be rewritten to a *latency graph* [34] to express these latency responses. For example, the LLR between two actors, s and d, can be observed as follows: Add an actor *src* with an edge to s with production rate q_s and consumption rate 1. Add an actor *snk* with an edge from d with production rate 1 and consumption rate q_d. If we let *src* fire once and wait until *snk* fires, we know the LLR from s to d. The ELR can be determined by, instead of having an edge from d to *snk*, replicating all edges coming into d, to also come into *snk*, i.e. by giving *snk* the same enabling conditions as d. In the remainder, we focus on LLR constraints.

Instead of producing a latency graph for each latency constraint and verifying them independently, the same idea can be applied to express latency constraints as used for throughput constraints [69]. A latency constraint between actor s and actor d implies that d limits the throughput of s. This is expressed by adding a dependency on d to s. This dependency consists of an added actor c, with an edge from d and an edge to s. The firing time of actor c is a function δ on the maximum allowed latency L. Below, we will explain why a meaningful latency constraint requires a constraint on the jitter of the arrival rate. This jitter is modelled by a parameter β. The dotted elements in figure 4.8 model latency constraint c with parameters β and L.

Of course, latency constraints should be formulated in application specific terms and not in terms of the SDF graph modelling the application. To this end, we offer the following intuition: A latency constraint specifies the maximum allowed time between the arrival of a unit of work and its LLR. However, since any platform offers only a finite throughput, the rate at which work arrives effects the latency per unit of work. Given that the first unit of work arrives at time t_0 then its LLR will occur no later than $t_0 + \lambda$, where λ depends on the application itself and the execution layout constructed for the application. When consecutive units of work arrive at the same rate as the running application's guaranteed throughput, all units of work will

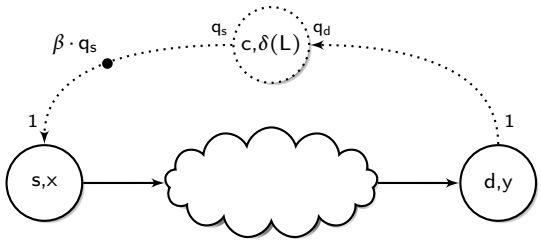


Figure 4.8 – Latency constraint expressed as throughput limitation

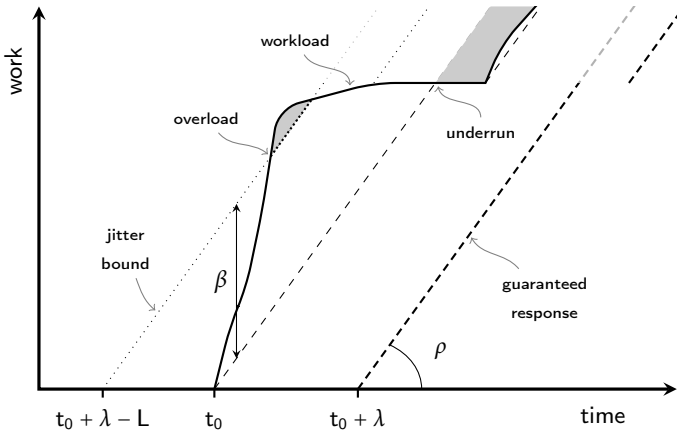


Figure 4.9 – Illustration of the parameters of latency constraints

incur a maximum latency of λ. If work arrives at a higher rate than the guaranteed throughput (overload), the latency per unit of work may increase. If work arrives at a lower rate than the guaranteed throughput (underrun), the maximum latency will still be λ, but the desired throughput will, of course, not be achieved. The arrival of work and the parameters of the guarantees are illustrated in figure 4.9.

To bound the latency per unit of work, given a throughput guarantee ρ, thus requires a characterization of the arrival rate. This characterization is given in terms of a maximum bound on the arrival jitter β: All work that arrives no later than $t_0 + t \cdot \rho$ and no sooner than $t_0 + t \cdot \rho - \frac{\beta}{\rho}$ will incur a latency of at most L.

Given a minimum throughput constraint ρ and constraint parameters β and L, the

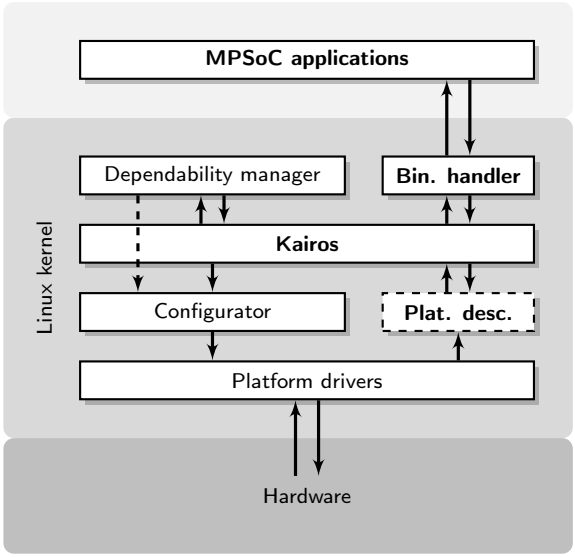


Figure 4.10 – CRISP software stack

firing time of actor c in figure figure 4.8 is defined as

$$\delta(L) = \max\left(\frac{\beta}{\rho} - L, 0\right)$$

4.5 Implementation: Kairos

In the context of the CRISP project [80], an on-line spatial resource manager in a Linux kernel is developed to manage the project's development platform. The platform contains one ARM processor (running the Linux kernel) and numerous other processors and managed memories (described in more detail in section 5.1.1) to which tasks can be assigned. The on-line spatial resource manager as described above and components required for its integration in a Linux kernel are implemented in a prototype implementation, called Kairos. Kairos requires a description of the platform (the processors and memories mentioned above and the connections between them). Furthermore, a handler is required to load binaries, i.e. the representation of applications on ternary storage. Figure 4.10 shows the CRISP software stack. The elements in bold are all part of the prototype implementation and discussed below. The other elements are required by the CRISP targets. To produce a system with run-time fault detection capabilities, the dependability manager periodically tests HwEs. The configurator performs run-time reconfiguration of the platform.

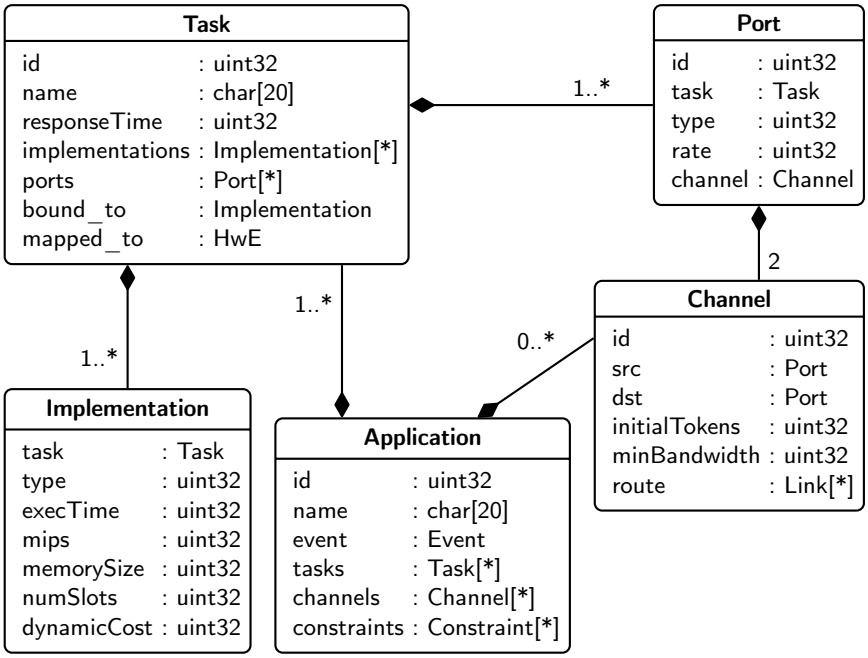


Figure 4.11 – Structures used for Kairos meta-data.

4.5.1 User interface: starting applications

Starting applications should, from a user's perspective, be 'business as usual'. On Linux, this means that a binary can be started from a terminal and the kernel handles the bootstrapping of the binary file to a running process. The type of application that can be run on some of the processors in the platform, rather than on the Arm running Linux, is different from 'normal' applications, because they can not be expressed in the common binary formats [74, sect. 3.12]. These MPSoC-applications must carry with them the meta-data required by Kairos to assign platform resources.

Data structures

The data structures Kairos uses for this meta-data can be modelled as shown in figure 4.11. All the meta-data described in this chapter, including the task graph and implementation set can be expressed in this data structure. The binary file format must contain a data structure as shown in figure 4.11.

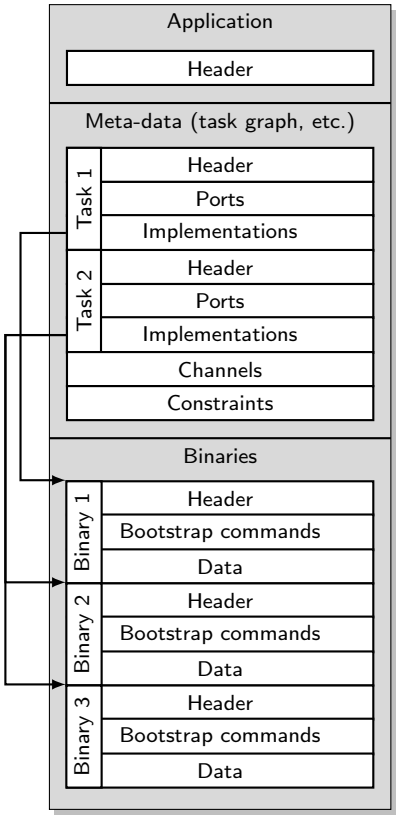


Figure 4.12 – The MPSoC binary file format

Binary file format

The MPSoC binary file format is inspired by Apple's Macho-O file format, which is used to store multiple (binary) programs and libraries in a single file, providing data and both statically and dynamically linked library code [5]. Similar to how Apple arranged their *universal binaries*, the MPSoC binary file format packs the different implementations for all the tasks of an applications into a single binary file. The file's header contains the meta-data.

Figure 4.12 shows the MPSoC binary file format. All meta-data is grouped at the beginning of the file, to prevent unnecessary seeking in the file. After Kairos has assigned tasks to implementations and resources, the configurator (see figure 4.10) can load only the required binary data from the file, without seeking, because the file offset is known from the implementations field of the task record. A more detailed specification of this file format is given in [15].

The file format corresponds closely to the data structures used by Kairos, which makes constructing the meta-data for Kairos very simple. This gives a straightforward and well-defined entry point into the kernel, which is discussed next.

4.5.2 Linux kernel workflow

A core task of an operating system kernel is the starting of programs. Regardless of the user interface, when a program must be started, the system call exec is invoked in the Linux kernel. At this point, the kernel does not load the program's binary file ('binary', for short) from disk, but uses the binary's header to find a handler that supports the file format of the binary. It does this by means of the search_binary_handler function. We introduce a new binary handler for the MPSoC binary file format.

From the file header described in figure 4.12, the data structures described in figure 4.11 are initialized. Next, they are placed in Kairos' request queue. Kairos uses a queue, because it can not allocate resources for multiple applications simultaniously. Using a queue also allows for the reordering of requests, e.g. releasing resources after an application terminates is a very cheap (fast) operation, so removal requests on the queue should always be handled before allocating resources to new applications.

When taken from the queue, a request is handled by Kairos. Upon completion, Kairos reports the result to the user and, if successful, passes the resulting execution layout to the platform configurator. This structure is shown in figure 4.13.

4.5.3 User interface: interaction with running applications

In accordance with Linux custom, Kairos makes information about running applications available in the Process File System (ProcFS). However, to list applications or tasks from applications in the normal process table is not possible. This would require the kernel to provide scheduling parameters and memory maps. Since the kernel is assumed to run on a controlling processor, but not on the HwEs, it can not provide these data.

Instead, Kairos creates custom entries in ProcFS that allow users to interact with applications running on the platform. Additionally the platform state and other useful information is exported to ProcFS. This allows the user to instruct the resource management algorithms (e.g. adjust configuration parameters) and to monitor the platform. For a full specification of the complete user interface, we refer to [15].

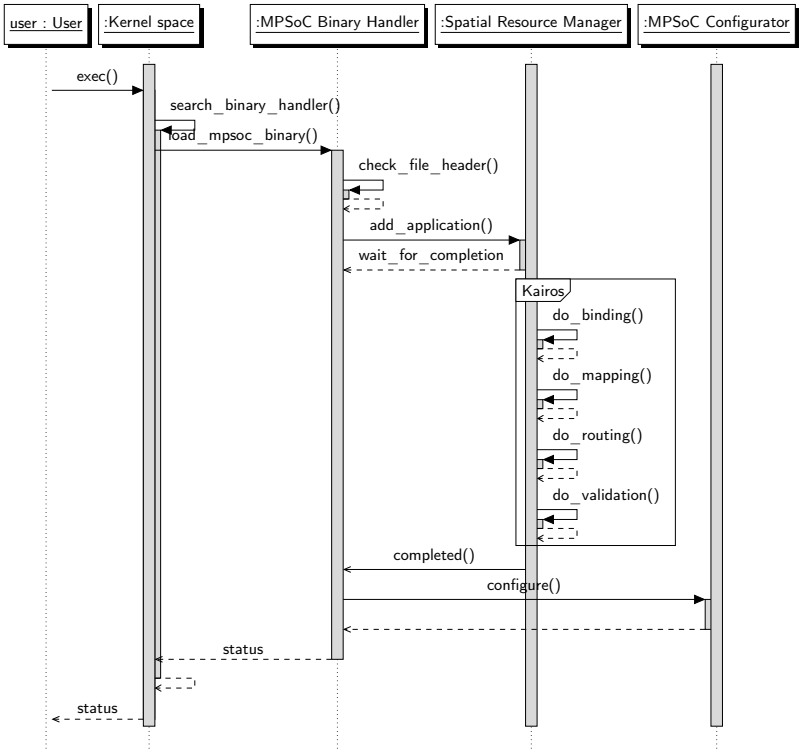
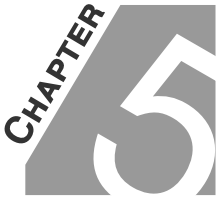


Figure 4.13 – Sequence diagram of Kairos' kernel workflow

4.6 Conclusion

In this chapter, we have shown how on-line spatial resource management can be implemented. This was done on a conceptual level by algorithms and their analysis, and on a concrete level by means of an implementation in a Linux kernel, providing user interfaces that adhere mostly to common practices. Furthermore, the input required by the on-line spatial resource manager has been specified and intuitions have been offered to designers of applications on how to interpret abstract concepts in this input.



OSRM EXPLORATION

ABSTRACT – *In this chapter, the algorithms introduced in the previous chapter and, more specifically, their implementations are applied in a case study and to a benchmark. The performance is evaluated in terms of the quality of the result and in terms of computational and memory cost. The case study is a concrete case from Thales, an industrial parter in a European project. The benchmark is developed for this purpose and discussed in detail before the results are presented.*

5.1 CASE STUDY: BEAMFORMER

Beamforming is a signal processing technique to control the directionality of a generated or received signal. One scenario in which it is applied is with *phased array antennas* [84, Ch. 9], where a set of small non-directional antennas simulate one large directional antenna. Without having to move the antenna array, the simulated antenna can be pointed towards a signal source electronically. When the non-directional antennas are arranged in a line or on a two-dimensional surface, a wave front coming into the array at an angle will arrive at the different antennas at different times. By delaying signals received by antennas, such that all signals are synchronised in time, and summing up the delayed results, signals from other directions are suppressed and the signal from the direction of the beam is amplified. Alternatively, beams can also be formed by operations in the frequency domain by shifting the signal phases of the antennas. This method is especially useful in digital

beamforming, since multiple beams can be formed from the same antenna array by applying different phase shift values to the same signals. In radar systems, for example, this is used to track multiple targets simultaneously. From an application point of view, this means there are as many input streams as there are antennas and as many output streams as there are beams being formed.

The beamforming application used for this case study was developed in the CRISP project [80]. It is a phased array radar application for nautical use. It is a streaming application, characterized by a high data rate and critical timing requirements. Some filtering and equalization is employed to improve the quality of the individual input streams, i.e. signals from antennas. The output streams, i.e. the beams, require Doppler analysis and other radar filtering and processing [84]. For the case study, 16 antennas and 8 beams were assumed.

The purpose of this case study is twofold. On the one hand, it tests Kairos to see how it performs with a single application with resource requirements close to the total resource capacity offered by the platform it must run on. On the other hand, it is used to demonstrate a real-life, large scale application. Both success rate and execution time are of importance for this test.

5.1.1 PLATFORM

The CRISP platform consists of three types of devices: Reconfigurable Device (RFD), General Purpose Device (GPD) and Field Programmable Gate Array (FPGA). A RFD contains nine Xentium processors[1], two smart memories and a dependability manager (used for run-time testing of processors, memories and interconnect). As described in chapter 3, every one of these components is seen as a HwE. The HwEs on a RFD are interconnected by a NoC. A GPD contains one ARM processor with memory and I/O-facilities for system control and observation. The CRISP platform contains five RFDs, one GPD and one FPGA. The GPD is used as a controller of the total platform, running a Linux kernel with Kairos. The interconnection of the GPD, RFDs and FPGA is realized by a point-to-point, address mapped interconnect with two ports on the GPD, four on each RFD and ten on the FPGA. A schematic representation of this platform is shown in figure 5.1.

5.1.2 APPLICATION

At the time of writing, no exact execution times for tasks are known, as the development of the application is still ongoing. However, preliminary numbers on processing load on HwEs are available. We use these numbers to derive an execution time for each task.

We assume that a Xentium processor running at 200 MHz has a processing power of 800 Million Multiply Accumulate (MMAC) per second. Tasks are assumed to

[1] A proprietary architecture, developed by CRISP participant Recore Systems B.V.

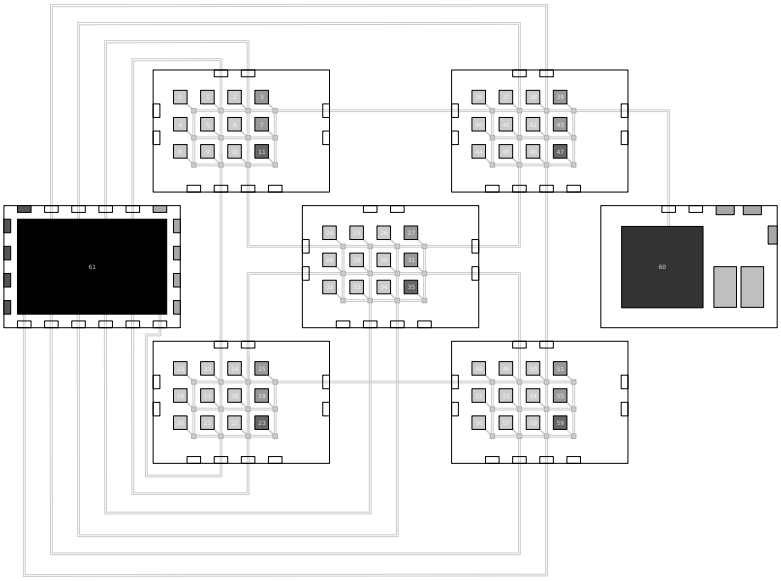


Figure 5.1 – Visualisation of the CRISP platform

impose a load of 640 MMAC per second at a token rate of 2.5 MHz. This renders an execution time per token of 320 nanoseconds:

$$\frac{640\frac{MMAC}{s}}{800\frac{MMAC}{s}\cdot 2.5MHz} = 320ns$$

This execution time per token is smaller than the clock period of the Xentium, because samples are processed in parallel. Since no data is available (yet) to model the application closer to the actual implementation, we except this inaccuracy. Modelling the system with such a sequentiality is a worst-case assumption, even if the time required per token in that sequential formulation is optimistic.

Every task of the beamformer has only one implementation. Therefore, this test case only tests the mapping, routing and validation phase. However, the implementation of Kairos is not changed for this problem, so the binding phase is executed. Thus, the effects caused by the binding phase's initialization of the mapping phase (see section 4.1.1) is still taken into account. Because Xentiums are not assumed to be multi-tasking in this platform, the execution times estimated by the method described above are considered to be response times. The whole application model is shown in figure 5.2, using the SDF notation introduced in chapter 4. The contents of these actors are not discussed here. They are irrelevant to the execution of Kairos. What is important is the structure of the application, the firing times of the actors and the token rates of the edges. The grey circles labelled pf_i are clusters of actors.

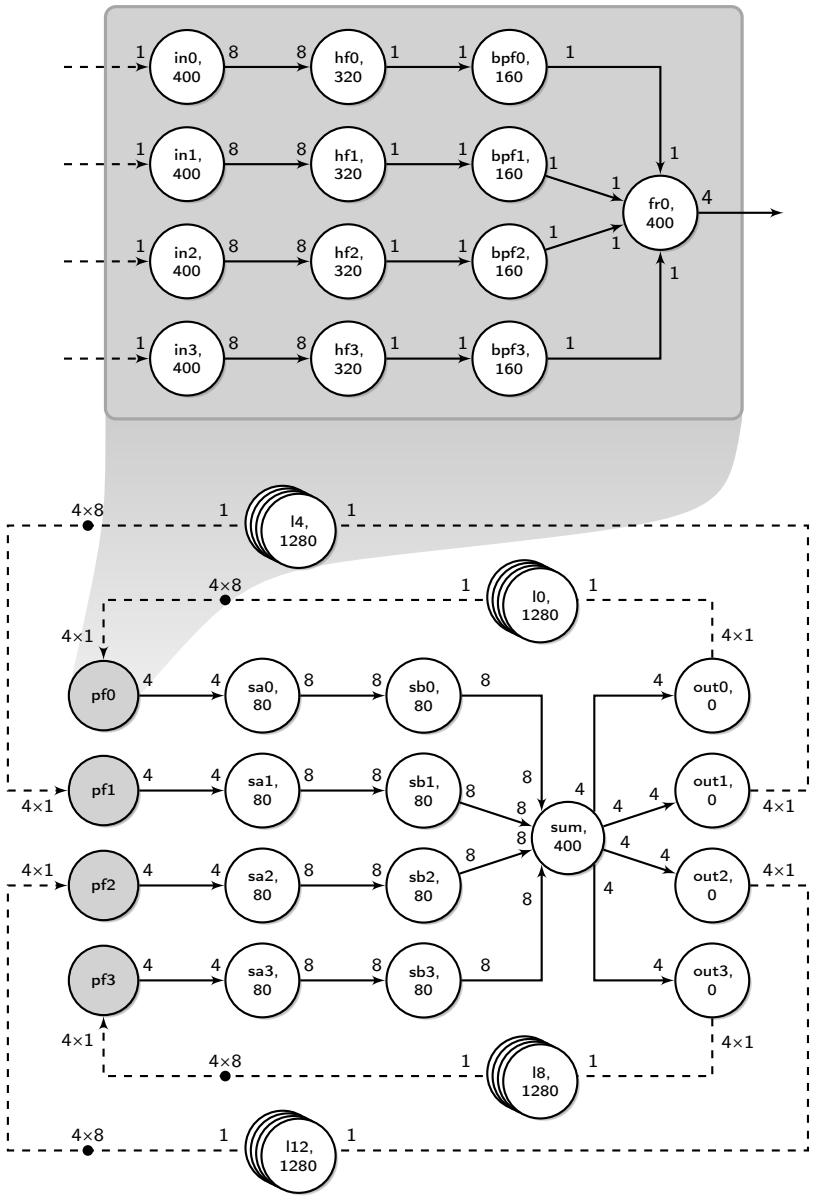


Figure 5.2 – Beamformer application model (buffers and self-loops omitted)

For brevity, only the expansion of pf_0 is shown.

Performance constraints

This application has a specified input frequency, from which the throughput constraint is derived. The input frequency is $2.5 \cdot 10^6 Hz$, thus the throughput requirement is:

$$\rho = 2.5 \cdot 10^{-3} \frac{tokens}{ns}$$

The application buffers are unknown, so we model buffers conservatively by assuming the smallest buffer sizes needed for deadlock-free execution. Added buffering from communication paths and the required synchronisation may cause significant extra latency. Therefore, latency constraints are required to guarantee the application's maximum latency specification. As described in section 4.4.4, a bound β on the input jitter is required. Even though the input is strictly periodic, we do specify $\beta = 8$ to allow the pipeline to be filled completely (i.e. maximally parallel execution).

The sum of response times of the application's critical path is 1840 ns. The lowest response time in the application is 80 ns. With assumed minimal buffer sizes, the maximum allowed LLR is $1840 + 80 = 1920\ ns$. The firing time of the latency constraint actors l_i is thus determined by

$$\delta(l_i) = \frac{\beta}{\rho} - l_i = 3200 - 1920 = 1280\ ns$$

For brevity, the buffer models and self-loops have been omitted and the (dashed) edges and actors modelling latency constraints have been grouped per four in figure 5.2.

5.1.3 RESULTS

The beamformer application requires I/O tasks to be fixed on specific ports of the FPGA, i.e. every input task is fixed to a specific port. In other words, the mapping of every I/O task is constrained to a specific FPGA port. When these mapping constraints are omitted, Kairos generally fails to find execution layouts. This is caused mainly by the segmentation of the platform. Starting from the FPGA, the platform appears homogeneous, but when tasks from the same clustered input (*pf i* in figure 5.2) are mapped onto different RFDs, either not all channels are mapped onto routes, or latency constraints are violated. The assumption of the fixed I/O tasks is reasonable, however, because in any final implementation, the input tasks are also bound to specific I/O modules. With these fixed starting points, the success rate is sensitive to the weights of the fragmentation and communication components of the cost function (see the discussion of the MapC algorithm in section 4.2). Figure 5.3 shows

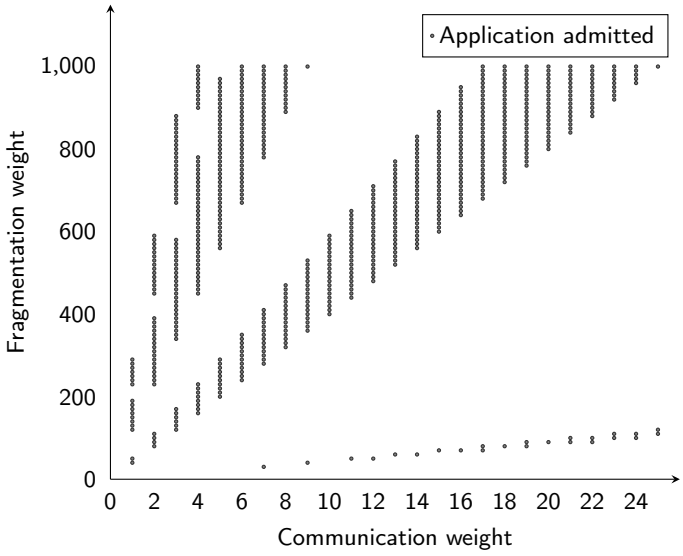


Figure 5.3 – Admission of the crisp application on the crisp51 platform with varying mapping parameters.

Cost function	Binding [ms]	Mapping [ms]	Routing [ms]	Validation [ms]
Communication only	57.39	83.19	54.91	12.64
Comm. & fragmentation	54.19	101.88	54.44	12.69

Table 5.1 – Kairos (averaged) run-times per phase

for which values of these weights Kairos admitted the application to the platform, i.e. Kairos found a feasible execution layout. All integer fragmentation weights between zero and one thousand and communication weights between zero and twenty five were tested. The fact that, for many combinations of fragmentation and communication weights, the application was not admitted demonstrates two things about Kairos. Firstly, Kairos is clearly sensitive to its configuration. Secondly, in this scenario, where there is little room in the resource budget and QoS constraints, Kairos does not guarantee success. Arguably for the second observation, however, the platform in this case study has clearly been designed with this application in mind. Thus, a lot of specific knowledge with regards to the (limitations of the) flexibility for spatial resource management that could be taken account has not been used to steer Kairos.

Table 5.1 shows the time used by Kairos to find an execution layout (or conclude failure). These run-times are averaged over a few executions, but the variance was near the reliable lower bound of the measurement itself. The run-times of the validation phase are notably low. This is mainly due to the very high degree of regularity in the application's task graph and the assumption of minimal buffers (making them proportional to the number of actors on the critical path). These run-times show that even for very large applications on irregular platforms, Kairos performs very well in terms of speed. This aspect of Kairos' performance is examined in more detail in the next section.

5.2 Synthetic benchmarks

To allow on-line spatial resource management, applications must be designed and implemented with some flexibility. Typically, their timing constraints should not approach the best possible performance of the platform on which they have to run, different implementations of tasks should be functionally interchangeable and the information required by Kairos must be available. The number of currently available real-life applications that meet these requirements is low and many of those applications are proprietary, i.e. their exact specification is not published. To compensate for this lack of test cases, we use a synthetic benchmark. For a few hand tailored platforms (discussed in section 5.2.1), artificial applications are generated and selected (section 5.2.2).

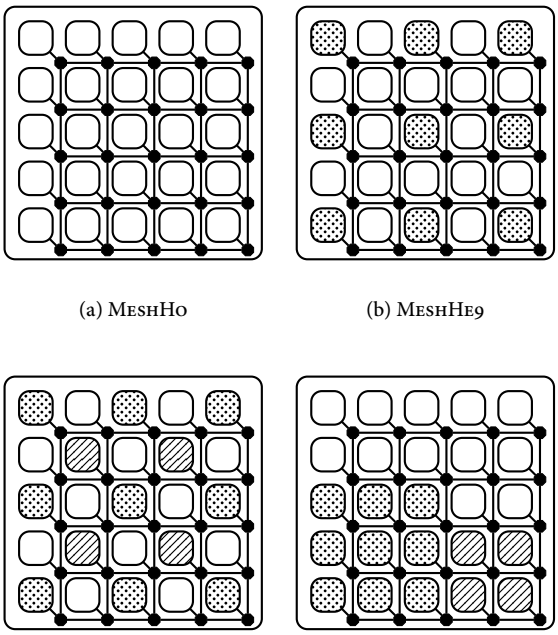
The applications used are typically an order of magnitude smaller (in terms of the number of tasks) than the platforms on which they have to run (in terms of the number of HwEs). These synthetic benchmarks are more representative for the general case in which on-line spatial resource management is to be applied.

The purpose of the synthetic benchmarks is to compare Kairos' run-times to those of reference (exact) solutions (section 5.2.3) and to compare the (cost of the) solutions found. Furthermore, the trade-offs to be made in application and platform design can be evaluated with good synthetic benchmarks.

5.2.1 platforms

The platforms used in the benchmark are designed such, that there is a clear difference in the degree of heterogeneity. Because the reference solutions do not take into account scheduling of HwEs, all HwEs in the benchmark platforms are single-tasking.

The platforms defined for the benchmark are shown in figure 5.4. The MeshHo platform (a) is a homogeneous 5×5 mesh of HwEs. All HwEs are of the same (abstract) type 'one'. The MeshHe9 platform (b) is derived from MeshHo, by replacing nine HwEs of type one by HwEs of type 'two' in a regular pattern, as



(a) MeshHo (b) MeshHe9

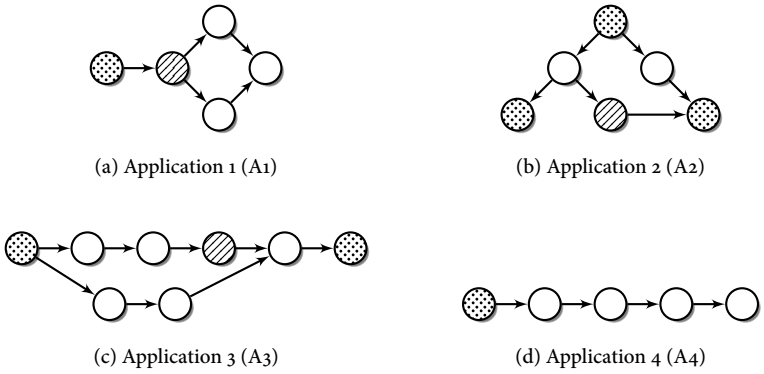
(c) MeshHe94 (d) MeshHeClust

Figure 5.4 – Platforms with varying degrees and distribution of heterogeneity

shown in dotted shading. A further regular replacement of four type one HwEs by HwEs of type 'three' transforms MeshHe9 into MeshHe94 (c). The last mesh platform, MeshHeClust (d) reorders the HwEs of MeshHe94 and is included to test the sensitivity of solutions to the spreading of heterogeneity. It effectively tests the assumptions with regards to the dominant factors in the hierarchical decomposition of the search process, i.e. when the reference solutions provide different bindings, the assumption that binding costs are of a considerably larger influence on the total cost than communication costs is shown to be wrong.

5.2.2 Application sets

Based on the ideas presented by Kolisch, et al. [56], we have developed an application generator. Task graphs produced by the generator are similar to those produced by the Task Graphs for Free generator [27], but they are annotated with one or more implementations per task, with the appropriate requirement vectors. Because the reference solutions can not perform the QoS validation, no SDF annotation is generated.



(a) Application 1 (A1)

(b) Application 2 (A2)

(c) Application 3 (A3)

(d) Application 4 (A4)

Figure 5.5 – Task graphs of the selected applications (shading indicates the availability of an alternative implementation)

From the generated applications, a selection is made, such that no more applications can be added without overloading the platform. The selection consists of four applications that together require slightly less resources than the platforms offer. The applications' task graphs are depicted in figure 5.5. All tasks in these applications have implementations for the basic (i.e. unshaded) HwE type. The shaded tasks also have an implementation for the HwE type drawn with the same shading in figure 5.4.

5.2.3 Reference solutions

In order to find reference solutions, we model (part of) the overall problem as an Integer Linear Program (ILP) (see below) and solve this by CPLEX [1]. Preferably, reference solutions should give a perfect solution of the overall problem. Unfortunately, our problem is too complex to be solved in pure ILP-terms, especially on the workstations and/or servers available to this research (most notably, combining spatial resource management with temporal scheduling is too complex). Although using more advance programming models (such as second-order cone programming [14]) on larger computer systems could likely provide more detailed solutions, we consider the likelihood that an exact solution for non-trivial cases in complete detail can be found in reasonable computation time to be very low.

Applications are assumed to arrive in-order. A perfect solution is a clairvoyant one, that solves the assignment problem for all applications at once. Whether or not to search for clairvoyant solutions does not significantly change the ILP, but rather the manner in which it is applied. Clairvoyant solutions can be found by providing all applications as input to the ILP at once, whereas applying the ILP incrementally to

the applications gives the in-order solution. The only difference in the ILP is that not all tasks have to be assigned, but rather either all or none of the tasks *per application* have to be assigned.

To reduce the complexity, two notable concessions are made in the formulated ILP. Firstly, the distinction between, on the one hand, routers and HwE and, on the other, interfaces and links is not modelled. In the ILP, a platform consists exclusively of HwEs and links. This concession does not harm the precision of the solutions found, because the platforms are modelled such that all (resource) restrictions imposed by routers and interfaces can be formulated in terms of restrictions imposed by links. To be able to route channels across multiple links, routers are modelled as HwEs with zero capacities, so that no tasks can be assigned to them, but links can be traversed between them. Secondly, no validation of execution layouts occurs, i.e. the QoS constraints are not checked. This concession *does* introduce imprecision in the solutions found: The resulting cost becomes a lower bound on the cost of the real optimal solution.

The ILP used to find reference solutions, discussed below, is the following:

$$\text{minimize} \quad \sum_{t\in T}\sum_{i\in I(t)}\sum_{e\in E}\zeta^{\pi}_{tie}\alpha^{\pi}_{tie} + \sum_{c\in C}\sum_{l\in L}\zeta^{\gamma}_{cl}\alpha^{\gamma}_{cl} \tag{1}$$

$$\text{subject to} \quad \sum_{i\in I}\sum_{e\in E}\alpha^{\pi}_{tie} = 1 \qquad \forall t \in T \tag{2}$$

$$\sum_{i\in I}\sum_{e\in E}(1-\tau_{ie})\alpha^{\pi}_{tie} = 0 \qquad \forall t \in T \tag{3}$$

$$\sum_{\substack{\langle u,v\rangle\in L\\ v=e}}\alpha^{\gamma}_{c\langle u,v\rangle} \le 1 \qquad \forall c \in C, e \in E \tag{4}$$

$$\sum_{\substack{\langle u,v\rangle\in L\\ v=e}}\alpha^{\gamma}_{\langle s,d\rangle\langle u,v\rangle} + \sum_{i\in I(s)}\alpha^{\pi}_{sie} = \sum_{\substack{\langle u,v\rangle\in L\\ u=e}}\alpha^{\gamma}_{\langle s,d\rangle\langle u,v\rangle} + \sum_{i\in I(d)}\alpha^{\pi}_{die} \qquad \forall \langle s,d\rangle \in C, e \in E \tag{5}$$

$$\sum_{t\in T}\sum_{i\in I(t)} r^{k}_{i}\alpha^{\pi}_{tie} \le c^{k}_{e} \qquad \forall k \in R, e \in E \tag{6}$$

$$\sum_{c\in C} r^{k}_{c}\alpha^{\gamma}_{cl} \le c^{k}_{l} \qquad \forall k \in R, l \in L \tag{7}$$

$$\alpha^{\pi}_{tie} \in \{0,1\} \qquad \forall t \in T, i \in I(t), e \in E \tag{8}$$

$$\alpha^{\gamma}_{cl} \in \{0,1\} \qquad \forall c \in C, l \in L \tag{9}$$

with input

ζ^{π}_{tie}	cost of assigning task t with implementation i to HwE e
ζ^{γ}_{cl}	cost of assigning channel c to link l
τ_{ie}	1 if implementation i and HwE e have the same type, else 0
r^{k}_{i}	component k of the requirement vector of implementation i

r_c^k component k of the requirement vector of channel c

c_e^k component k of the capacity vector of HwE e

c_l^k component k of the capacity vector of link l

and output

α_{tie}^{π} whether task t is assigned to HwE e with implementation i

α_{cl}^{y} whether channel c is assigned to link l

The objective (eq. 1) is to minimize the total cost of all assignments. With regards to the task assignments (α_{tie}^{π}), all tasks must be assigned precisely once (eq. 2). Furthermore, it must hold that no implementation is assigned to an HwE of another type (eq. 3). Communication channels should not be mapped onto cyclic paths. To prevent this, eq. 4 states that any channel may arrive at any one HwE only once (with routers modelled as HwEs). The balance equation eq. 5 states that any channel that comes into an HwE must either end there, or also go out (and vice versa). Finally, the cumulative resource requirements of either tasks or channels may not exceed the capacities of the HwEs and links they are assigned to (eqs. (6) and (7)) and all assignments are binary (eqs. (8) and (9)).

5.2.4 RESULTS

The benchmark results are presented in full detail in appendix A. In this section, we first evaluate the performance of Kairos, compared to that of the reference solution. Because the case study in section 5.1 shows that Kairos is sensitive to its configuration (the balance between the fragmentation and communication components of the cost function), four configurations of Kairos are tried. One configuration tries to balance communication and fragmentation. Two configurations try to optimize for one of the two. The last configuration adds no cost for either. The details of these configurations are also in appendix A. Performance is measured both in terms of the quality of the result, as well as in the time and memory required to produce it. Next, we discuss observations made during these benchmarks with regards to platform design.

Resource management methods

It is important to note that Kairos performs QoS validation, whereas the reference solution does not. This means that the cumulative costs of the reference solution are lower bounds (i.e. not all solutions found by the reference solution are guaranteed to be feasible). Also, Kairos' requirements, both computational and memory, are significantly lower than those of the reference solution.

Cumulative cost Figure 5.6 shows the summary of the cumulative costs. For all four architectures (shown in figure 5.4), the applications (A1 through A4, shown in

figure 5.5) are started, one after the other. Then, the platform is reset (the vertical lines through the x-axis in the figure) and the applications are started in reverse order (A4 through A1). The graph shows the sum of costs all running applications, where the striped bars represent the results of the incremental ILP, and the dotted bars show the results of the different Kairos configurations: The dotted area shows the best resulting cost, the top of the unshaded area shows the worst and the grey line indicates the average of all four Kairos configurations. Quite remarkably, for MeshHeClust, the incremental ILP failed to find a solution for application A4 after the other three applications were started.

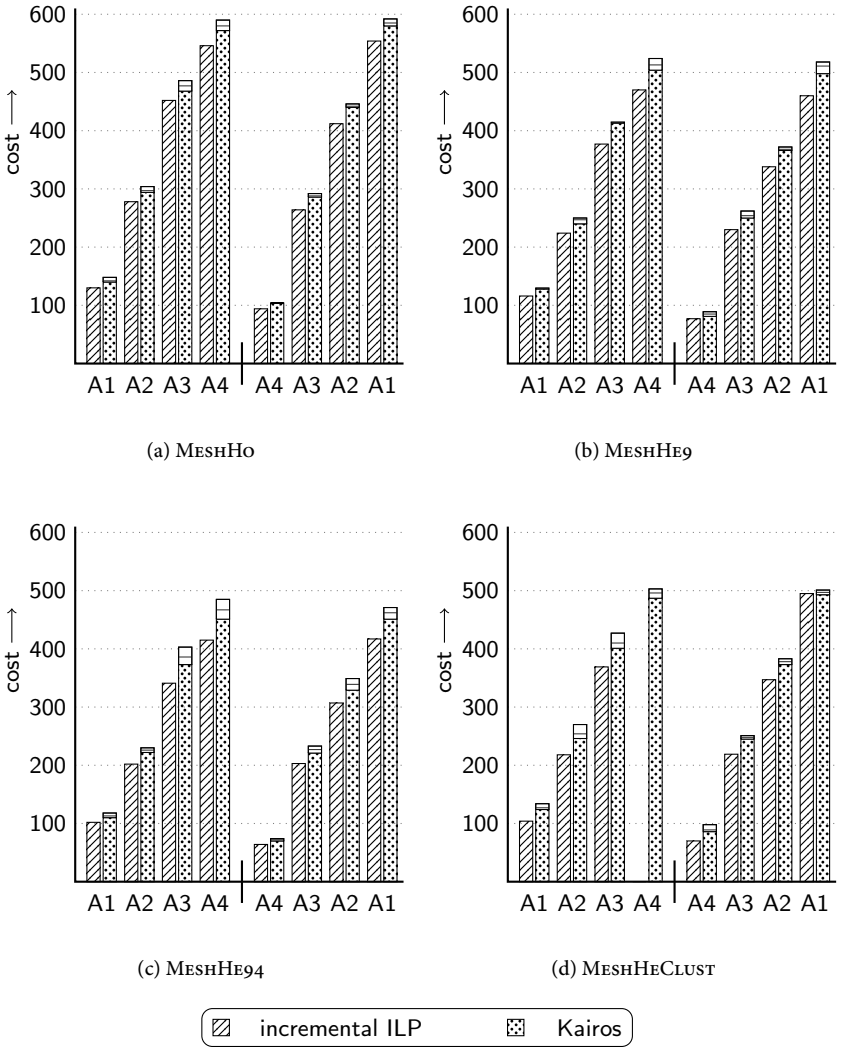
On average, the cost performance of Kairos is approximately 11% above that of the reference solution. The relative difference typically reduces as more applications are added to a platform. The average result over the different Kairos implementations is further away from the ILP solution for increasing heterogeneity of the platforms. We ascribe this sensitivity to the separation of binding and mapping, because binding does not take locality of available HwEs into account.

The fact that there is no solution for application A4 on platform MeshHeClust after the incremental ILP has started applications A1 through A3, is a lucky coincidence for Kairos in this benchmark. However, it does demonstrate that using the optimal assignment for every individual application, does not guarantee an overall optimum. This case did beg for a clairvoyant ILP solution, where time is modelled in intervals and the arrival of applications is known beforehand. The results of this clairvoyant ILP are listed in appendix A. However, since scenarios in which the arrival order of applications is known beforehand can be scheduled completely off-line, clairvoyant solutions have only weak relevance to the problem at hand.

Performance Kairos is one order of magnitude faster than the reference solution in the worst case (all applications in all scenarios were finished in under 50 ms) on a core (Arm926 at 200 MHz), that is at least one order of magnitude slower than a single core (Intel Xeon E5430 at 2.66 GHz) on which the ILP solution is executed. Some solutions for the incremental ILP took more then ten seconds to find. The worst-case clairvoyant solution (homogeneous platform) took over 86 hours. Furthermore, the ILP solution is executed on a *quad-core* processor (that are all under near full load throughout the execution), whereas Kairos is executed on a single core. It is thus safe to conclude that Kairos is at least two orders of magnitude faster than the ILP solution and, in many cases, considerably more.

The difference in memory usage is also considerable. The incremental ILP solution reported memory usage of 80–90MB. Unfortunately, it is hard to measure the memory usage of Kairos accurately. The problem is that Kairos is not a separate process, but part of the kernel. The Linux kernel improves its own performance by caching requested memory pages. We have analysed the size of the kernel cache after starting every application[2]. The maximum kernel memory usage, including all

[2] Cache sizes were taken from the `/proc/slabinfo` facility.



(a) MeshHo

(b) MeshHe9

(c) MeshHe94

(d) MeshHeClust

Figure 5.6 – Cumulative cost

cached pages never exceeded 25MB.

Platforms

Since the execution costs of specialized implementations are lower than those of their counterparts for general purpose HwEs, it follows that the cumulative costs are lower as well. However, these benchmarks show that this heterogeneity is exploitable in practice. Platforms such as the one used in [68] rely heavily on resource management that can cluster communicating tasks on the same HwEs, because they are bandwidth constrained. Such clustering is not generally applicable when neighbouring tasks in a task graph do not have implementations for the same type of HwE. When locality can not be exploited by clustering, such platforms become bandwidth constrained. This observation can also be made about the different results for MeshHe94 and MeshHeClust, where for the former, results are better than for the latter. The fact that the ILP does not find a result for application A4 on MeshHeClust in one scenario is, because the trade-off between cheaper execution on a specialized HwE and the distance between that HwE and other HwEs to which other tasks of the same application are already mapped, is decided in favour of locality for some tasks in the first three applications. Starting application A4 fails, because there are not enough general purpose HwEs (the type of which the entire MeshHo platform consists of and for which every task has an implementation) are available. The observation that the exploitation of heterogeneity requires more interconnection capacity is supported by the measured average path length (see appendix A).

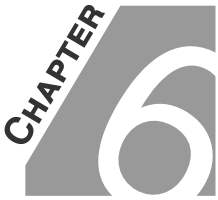
5.3 Conclusion

In this chapter, we demonstrated that Kairos is applicable in realistic scenarios. The time it requires to find execution layouts is well within the acceptable limits for the intended platforms and applications. The solutions found compare relatively well to exhaustively optimized solutions.

Kairos' performance (both in computational terms, as well as in terms of the result) improves significantly when applications are (at least) an order of magnitude smaller than the platform on which they have to run. Heterogeneity helps to reduce the execution cost of applications, both in low and high utilization cases. On-line spatial resource management can deal with heterogeneity, but if such flexibility is desired, the design of platforms should take into consideration some additional constraints, i.e. heterogeneity is best distributed evenly over the platform and the interconnect should offer sufficient resources.

Part II

Asynchronous Dataflow



Denotational semantics of SNet

Abstract – *This chapter offers a brief introduction to SNet, before discussing a denotational semantics of SNet. The semantics are expressed by means of a translation from SNet to Haskell. Non-determinism of SNet networks is expressed as an oracle, which is an argument to the function produced by the translation of the network. Using this translation, a proof is given that, for every oracle, every SNet network is prefix monotonic.*

6.1 Motivation

In this chapter, the denotational semantics of SNet is discussed. In the words of Scott and Strachey [83], the purpose of a denotational semantics is to "give a correct and meaningful correspondence between programs and mathematical entities in a way that is entirely independent of an implementation." Practical applications of a denotational semantics include proof of equivalence of two different programs (e.g. for semantics preserving compiler transformations) and reasoning about properties of the output of a program, given properties of the input of that program. These applications are the motivation for a denotational semantics of SNet.

The denotational semantics presented in section 6.5 are used to show that SNet is *prefix monotonic* in section 6.6. Prefix monotonicity is the specialized term for *causality* in streaming languages. The property of prefix monotonicity is important,

Parts of an earlier revision of this chapter have been published in [PhH:1].

because it enables the SNet programmer to make observations about networks, given partial input data. A prefix monotonic program is guaranteed to be consistent, i.e. future input does not change past output. Therefore, in terms of the position of an input element in the input sequence, it can be determined *precisely* when each of the output elements is produced. Consequently, a schedule can be derived for the execution of a prefix monotonic program on finite resources.

6.2 A brief overview of SNet

In this section, we introduce essential features of the language SNet, a declarative coordination language for asynchronous stream programming, to serve as background information. We deliberately exclude key language features, like type inference, that are not directly related to the subject of the work presented in this thesis. Moreover, the syntax used here is enriched with some shorthand notations, with regards to the official SNet syntax. For a complete coverage of SNet, including the official syntax, we refer the interested reader to [37].

6.2.1 networks, records and streams

Every network in SNet is siso. This means that every network transforms an input stream to an output stream. A stream is a (potentially infinite) sequence of non-overlapping, discrete data items, called *records*. The basic networks are *primitive networks* that can be combined by using *network combinators* into (non-primitive, siso) networks.

Primitive networks perform either *processing* or *synchronization*. Processing networks are stateless functions, defined by the user in one of two possible ways: A *box*, implemented in a programming language (referred to as the *box language*), or a *filter*, specified in SNet terms. Synchronization networks, known as *synchrocells*, combine records based on their type. All three forms of primitive networks are described in more detail in section 6.2.4.

Records contain *fields* and *tags*. Both of these are named. Fields contain values that are opaque to SNet, i.e. they are values for the box language. SNet can only rename, copy or delete fields, but any other transformation on fields occurs in boxes only. Tags contain integer values and are readable and writeable in SNet.

6.2.2 types, type matching and routing

Records are routed through a network dependent on their type. A *record type* is specified by the set of names of the fields and tags contained in records of that type. A subtype relation on record types is defined as the superset relation on sets, i.e. when record type t contains strictly more names than record type t', then t is a

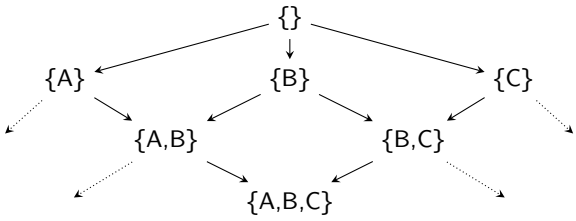


Figure 6.1 – Subtype relation between record types

subtype of t'. Figure 6.1 illustrates the subtype relation. A, B and C are the names of fields or tags. The universal supertype is the empty set. Every set with precisely one name is a subtype of the empty set. Every set with two names is a subtype of both sets that contain only one of those names. The subtype relation is transitive, e.g. (from the example in figure 6.1) $\{A, B\}$ is a subtype of $\{B\}$, which is a subtype of $\{\}$, so $\{A, B\}$ is a subtype of $\{\}$ also.

A network takes records from its input stream and results in records on its output stream. Thus, *network types* are defined in terms of record types. Networks take records of one type and result in zero-or-more records of possibly different types. Networks can take different types of records as input. These are referred to as *input variants*. The types of the records on a network's output stream depend on the input variant and (often) also on the *values* of fields and tags in the corresponding records on the network's input stream. Thus, for every input variant, a set of *output variants* (record types of records that the network can produce in response to the input) is given. Formally, if $\mathfrak{R}$ denotes the set of all possible record types, the set of all network types (denoted $\mathfrak{N}$) is defined as $\mathfrak{N} = \wp(\mathfrak{R} \times \wp\mathfrak{R})$, where $\wp$ denotes the power set. However, for the work presented in this thesis, only the input type of a network (the set of all the network's input variants) is important, because these types are used for the routing of records within the networks. For a detailed discussion of the SNet type system and inference of network types, see [17].

If records can be used as input for a number of different networks—e.g. in case of parallel composition, discussed below—records are routed into the network with the *strongest matching type*. Here, a record type t_r matches a network type t_n when t_n contains at least one input variant t_v that is equal to or a supertype (i.e. subset) of t_r. This means that a network does not need all fields and tags in a record. In the next section, we discuss how the fields and tags not handled by the network are taken into account. The *strength* with which t_r matches input variant t_v is the size of t_v, i.e. the number of names in t_v. The strength with which t_r matches network type t_n is that of the strongest matching input variant of t_n.

6.2.3 FLOW INHERITANCE

As mentioned, records can be of a subtype of an input variant of a network. In such a case, the network does not affect the fields and tags with names not in the input variant. The fields and tags that are not indicated in the input variant of the network are said to 'flow around' the network. This is known formally as *flow inheritance*.

For example, consider a network with only one input variant $\{A, B\}$ and a record with type $\{A, B, C, D\}$. The record can be fed into the network, because its type is a subtype of the network's input variant. However, because the network is only defined to work on the fields (or tags) with the names A and B, only those fields are inserted. In other words, the record is split into a *through-* and an *around*-part. The through-part is a record $\{A, B\}$ and the around-part is a record with type $\{C, D\}$. The through-part is inserted into the network, in response to which a result is produced. The around-part is combined with the result. If the result is a record of type $\{X, Y\}$, then the combination of the result and the around-part is a record of type $\{C, D, X, Y\}$. When the result contains fields with names that also occur in the around-part, the fields in the result are preferred. Thus, if the result is a record of type $\{C, X\}$, the combined record has the type $\{C, D, X\}$ and the value of the field labelled C is that of field C from the result of the network.

Flow inheritance is only defined on *primitive* networks. Intuitively, many non-primitive networks behave as if flow inheritance was defined on them as well. However, such an intuitive understanding is often very misleading. The primitive networks and their behaviour with regards to flow inheritance is discussed in section 6.2.4. Section 6.2.5 describes the network combinators. For a more elaborate discussion on flow inheritance and its consequences on combined networks, see [17].

6.2.4 PRIMITIVE NETWORKS

As discussed above, SNet distinguishes three different forms of primitive networks: boxes, filters and synchrocells. The primitive networks are discussed in more detail in this section.

Box

An SNet box is a stateless user-defined processing primitive network. It is connected to the outside world via a single input stream and a single output stream. A box can start processing as soon as a record appears on its input stream. The concrete behaviour of the box is specified outside SNet using an appropriate *box language*. A box may emit zero or more records on the output stream in response to a record on the input stream; the exact number depends both on the box' implementation and

the values of the incoming record. We call this dynamic behaviour the *multiplicity* of the box.

Every execution of a box takes exactly one record from the input, i.e. multiplicity only occurs on the box' output. The entirety of a box' output after consuming a single record is called the box' *response* to that record. Furthermore, since the box is stateless, the response of the box depends exclusively on the single record of the input. Because a box is applied in-order to the records on its input stream, it carries over the order of the records on its input stream into a *causal order* on the output stream. In other words, causal order means, that if record a precedes record b on the output stream, one of two cases hold: Either the two are part of the same response to a record on the input stream *or* the record on the input stream to which a was (part of) the response preceded the record to which b was (part of) the response.

Flow inheritance for boxes with multiplicity is defined as follows. The around-part of an incoming record is combined (as described above) with *each* record in the box' response to the through-part. For example, assume a box that accepts records of type $\{A, B\}$ and produces records either of type $\{X, Y\}$ or of type $\{C, X\}$. Now assume an input record with type $\{A, B, C, D\}$, which is a subtype of the box' only input variant. The input record is split into a through-part, having type $\{A, B\}$ and an around-part, having type $\{C, D\}$. The through-part is consumed by the box. In response, the box produces two records: r of type $\{X, Y\}$ and r' of type $\{C, X\}$. The around-part is combined with both results. For r this results in a record of type $\{A, B, X, Y\}$. However, r' contains a field or tag with the same name C is a field or tag in the around-part. Thus, combining r' with the around-part, discards the C field or tag from the latter. In other words, C and X are taken from r' and D from the around-part to come to the combined result with type $\{C, D, X\}$.

Filter

A filter is similar to a box in terms of its behaviour as a processing primitive network on a stream, including causal order, statelessness, multiplicity and flow inheritance. It is different, because it is specified in SNet by means of (simple) expressions and it can not access fields (i.e. box language values). The structural representation of SNet's expression language for use in this thesis is described in section B.2.

Synchrocell

The purpose of synchrocells is to combine two or more records into a single record, based on their respective types. A synchrocell is defined by a list of record types. For each record type, there is a corresponding 'slot' in the synchrocell. In every slot, one record of that type can be stored. When a single record is stored in a slot, that slot is *filled*. Before that time, the slot is *free*.

Like boxes and filters, a synchrocell takes records from its input stream in-order and one at a time. Every incoming record is matched against those record types that correspond to free slots. For every record type of the synchrocell that the incoming record matches, the part of the incoming record corresponding to the record type of the synchrocell is stored in the corresponding slot, i.e. the through-part of the record with regards to the record type is stored. When an incoming record fills all remaining free slots, the synchrocell *syncs*: The records stored in the slots of the synchrocell are combined into a single record. This combined record is produced on the output.

This leads to three distinguishable scenarios that describe the synchrocell's behaviour: 1) If the incoming record matches the record types of all remaining free slots, the record resulting from the combination of all stored records is produced on the output. 2) If the incoming record matches at least one record type corresponding to a free slot (but not all remaining), nothing is produced. 3) The incoming record is produced 'as is' when it does not match any remaining free slot's record type. Thus, multiplicity occurs also for synchrocells: The response to a record is either one record or nothing. Finally, the slots of the synchrocell are *not* freed when the synchrocell syncs. This implies that, after syncing, a synchrocell passes through all incoming records (scenario 3).

Flow inheritance on synchrocells is defined slightly differently than on boxes and filters: The around-part of the (first) record that matches the first record type (in order of the specification) of the synchrocell is merged with the result when the synchrocell syncs. The around-parts of the other records are discarded. This behaviour is an SNet design choice.

6.2.5 SNet network combinators

SNet primitive networks, i.e. boxes, filters and synchrocells, are combined into (acyclic, siso) streaming networks by means of network combinators. In other words, network construction is an inductive process starting with boxes, filters and synchrocells as basis. In the following, different network combinators are explained.

Sequential composition

Given two networks N and M, the sequential composition of N and M (denoted N .. M) yields a network where all input is streamed into N, the output stream of N becomes the input stream of M and the output stream of M becomes the output stream of the combined network.

In principle, the sequential composition is similar to function composition. Sequential composition naturally preserves the causal order.

Sequential composition is associative. For brevity, we can thus write

$$\bullet\!\bullet_{N \leftarrow L} N$$

to represent the sequential composition of a finite list of networks L.

Parallel composition

Given two networks N and M and a boolean value δ, the parallel composition of N and M (denoted $N \parallel_\delta M$) is a network where the input stream is split up into two streams that are fed into N and M, respectively. More precisely, records on the input stream are routed to the network with the strongest matching type. If there are multiple networks with equally strong matching types, a non-deterministic choice is made among those networks. The individual output streams of N and M are merged to form the output stream of the new network. The merger is either deterministic (when $\delta = \mathsf{T}$) or non-deterministic (when $\delta = \mathsf{F}$).

Non-deterministically merging two streams means that records on those streams are non-deterministically *interleaved*. However, when record a appears before record b in the output stream of N, a also appears before b in the merged output stream. The non-deterministic merger of the two output streams of N and M does not preserve the causal order on the stream, since data being processed by N may well be overtaken by data processed by M or vice versa. The deterministic merger *does* preserve the causal order along both substreams.

Parallel composition is also associative. For brevity, we can thus write

$$\Big\|_{\delta,\, N \leftarrow L} N$$

to represent the parallel composition of a finite list of networks L.

Serial replication

The serial replication of a network N means, that every record in the output stream of N may be fed through another copy of N, until it matches a *terminator*. A terminator is a set of *patterns* (see [37] and, for the structural definition, section B.2). A pattern consists either of a single record type to be matched or of a record type and a *guard expression*. Guard expressions are expressions of tag values. Guard expressions have a boolean result. A record matches a pattern if it matches the pattern's record type and, if the pattern has a guard expression, the pattern's guard expression evaluates to *true*. A record matches a terminator if it matches any of the terminator's patterns.

Given a network N, a boolean value δ and a terminator γ, $N^\star_{\delta,\gamma}$ denotes the serial replication of N, which is itself a SISO network. We can interpret γ as a function

on records, i.e. $\gamma : \mathcal{R} \to \{recurse, done\}$, where $\mathcal{R}$ denotes the set of all possible records. Thus, the application of γ to a record a, i.e. $\gamma(a)$, is the matching of a against the patterns of γ. If any pattern matches, the result of this function is *done*, otherwise, it is *recurse*.

As before, there are two variants: deterministic ($\delta = \mathsf{T}$) and non-deterministic ($\delta = \mathsf{F}$). The deterministic variant preserves the causal order of the input stream in the overall output stream of the replicated network, whereas the non-deterministic variant does not (necessarily). The serial replication combinator can be understood recursively as:

$$N^{\star}_{\delta,\gamma} \approxeq (N \mathbin{..} N^{\star}_{\delta,\gamma}) \parallel^{\gamma}_{\delta} Id$$

where *Id* is the 'identity network' or 'bypass': Every record that comes into *Id* comes out of *Id*, unchanged and in-order. The superscript of γ on the parallel combinators indicates that records are not routed by the strongest match, but by any match (regardless of strength) of the terminator.

Inspection composition

Given a network N, a boolean value δ and the name of a tag t, the inspection composition of N inspecting t (denoted $N \mathbin{!}_{\delta} t$) is a potentially infinite parallel composition of N with itself, where incoming records are routed based on the *value* of tag t. In other words, for every value of tag t, a separate instance of N exists to which all records that hold the corresponding tag value are routed. This composition can be thought of as an equivalent to the switch-/case-statement found in many programming languages. As before, there are two variants: deterministic ($\delta = \mathsf{T}$) and non-deterministic ($\delta = \mathsf{F}$).

6.3 Purpose and approach

The purpose of the denotational semantics of SNet is to allow formal reasoning about SNet programs and with SNet programs about input streams, independent of the underlying execution model. An added benefit of a denotational semantics is that it offers a contract to which execution models must adhere. The approach taken to come to a denotational semantics of SNet is to offer a translation scheme from SNet to the functional programming language Haskell [79]. Haskell has a denotational semantics [2]. Haskell offers a wealth of techniques and tools—monads, arrows, etc.—but for the sake of accessibility, we use only basic constructs. Thus, the offered translation scheme can be understood in terms of any (pure and lazy) functional programming language.

Because SNet allows for (explicit) non-determinism, a translation to Haskell must bound non-determinism. In other words, because a pure functional program is

fully deterministic, SNet's non-determinism must be modelled in pure functional terms. We model non-determinism by an *oracle*. Any network containing non-determinism is translated to a function, that takes an oracle as an argument. Effectively, this brings all non-determinism "outside," i.e. transforming non-deterministic choices to parameters. By translating networks to pure functions with oracles, using the translation scheme described in section 6.5, formal reasoning using quantification over all possible oracles can be achieved. In section 6.6, this strategy is used to show that any SNet program is prefix monotonic.

The representation of SNet language elements—combinators, actions, types, etc.—in Haskell is described in appendix B. All Haskell code in this chapter has been formatted for brevity and readability. The corresponding string substitutions are listed in appendix C.

6.4 Data structures and utilities

In this section, we introduce the Haskell data structures and utilities we use in the translation from SNet networks to Haskell functions (see section 6.5). As utilities, we identify a few common patterns that occur in the SNet translation and express them in functions. The proof of SNet's prefix monotonicity (see section 6.6) is made easier this way: When we prove properties of a utility function, those properties are given for SNet translations expressed in terms of these utility functions.

6.4.1 types and evaluables

A detailed description of the representation of records, networks and types is given in section B.1. A few elements of that description are of importance for the understanding of this chapter. Therefore, we summarize them here.

Records ($\mathcal{R}$) have a record type ($\mathfrak{R}$) and networks ($\mathcal{N}$) have a network type ($\mathfrak{N}$). For both records and networks, a function is provided to derive the corresponding type. The matching strength of a record against a network type can be determined with the function `match`.

$$\begin{aligned}&\texttt{typeOf} &&:: \mathcal{R} \to \mathfrak{R}\\ &\texttt{netType} &&:: \mathcal{N} \to \mathfrak{N}\\ &\texttt{match} &&:: \mathcal{R} \to \mathfrak{N} \to \mathbb{Z}\end{aligned}$$

For flow inheritence, we define an inheritence structure ($\mathfrak{I}$) as having a `through` and an `around` part.

$$\texttt{data}\ \mathfrak{I} \triangleq \mathcal{I}\,\{\ \texttt{through}, \texttt{around} :: \mathcal{R}\ \}$$

We define two operators for flow inheritence: The separation operator ($\prec$) and the fusion operator ($\succ$). The separation operator takes as operands type *t* and record *r*

and separates r into an inheritance structure, where the through-part has type t (precisely t, i.e. not a subtype) and the around-part has everything in r that is *not* in the through-part. Network types are inferred from the types of the primitive networks of which they consist [37]. The types of these primitive networks are more specialized than network types. Boxes have box types ($\mathfrak{B}$) and filters have simple network types ($\mathfrak{F}$), i.e. network types with only one input variant. The separation operator is defined in the class `Inheritable`, of which box types, simple network types and record types are all instances. The fusion operator takes as operands an inheritance structure and a record and produces a record. It merges the around-part of the inheritance structure with the record, such that if they contain fields or tags with the same name, the values in the record are preserved and the values in the inheritance structure are discarded.

For example, let record r be input for a box with type t. Inheritance structure i is defined as $i = t \prec r$. The box is applied to the through-part of i, resulting in the (multiplicitous) result r'_1, r'_2. The around-part of i is combined with the result by applying the fusion operator to i and both records, i.e. $i \succ r'_1, i \succ r'_2$.

```
(≺) :: Inheritable t ⇒ t → ℛ → ℐ
(≻) :: ℐ → ℛ → ℛ
```

SNet also includes a few 'evaluable' language constructs. Filters consist of filter actions ($\mathfrak{A}$) and both serial replication combinators and synchrocells are defined using patterns ($\mathfrak{P}$). Both actions and patterns consist of expressions ($\mathfrak{E}$). All of these constructs can be evaluated, given a record as context, by the function `eval`.

```
eval :: Evaluable a ⇒ ℛ → a t → t
```

If a pattern matches the context record, it evaluates to a record type. When an action is evaluated with the context record, it results in a list of records. For the precise definition of these language constructs, see section B.2.

6.4.2 STREAMS

Intuitively, a stream is a list of records. However, this leads to a problem with observability. Consider, for example, the deterministic parallel composition of the networks N and M, i.e. $N \parallel_{\mathsf{T}} M$. Let the input stream of this network $in_{N \parallel_{\mathsf{T}} M}$ be given as

$$in_{N \parallel_{\mathsf{T}} M} = a, b, c, \ldots$$

That is, a, b and c are the first three records of the input stream. Furthermore, let the record types of a and c match the network type of N and not that of M, whereas the record type of b matches the network type of M and not that of N. Thus, the individual input streams of the networks N and M are

$$in_N = a, c, \ldots$$
$$in_M = b, \ldots$$

The problem of observability becomes apparent when looking at the output streams:

$$out_N = n, n', \ldots$$
$$out_M = m, m', m'', \ldots$$

It is unclear how these streams must be merged to form $out_{N \parallel_{\mathsf{T}} M}$. The preservation of causal order dictates that the responses to b succeed responses to a and precede responses to c. However, the correspondence between records on the output streams of N and M to those on the respective input streams is lost: Because of multiplicity, we can not assume that n is a response to a and n' is a response to c.

To solve this problem, we introduce delimiters in the stream. A delimiter is transparent to primitive networks, since primitive networks operate on records, one at a time and in-order. For all networks, we say that if there are k delimiters between records a and b on a network's input stream, there are k delimiters between the responses to a and b on the network's output stream. To illustrate, we revisit the example above. Let $\square$ denote a delimiter in the stream. We introduce delimiters between every record when splitting the stream and feed the resulting streams into the respective networks, i.e.

$$in_{N \parallel_{\mathsf{T}} M} = a, b, c, \ldots$$
$$in_N = a \square c \square \ldots$$
$$in_M = b \square \ldots$$
$$out_N = n, n' \square \square \ldots$$
$$out_M = m, m', m'' \square \ldots$$

In this example, the response to a are the records n and n', the response to c is empty and the response to b are the records m through m'' (recall that primitive networks can be multiplicitous). The correspondence is now observable, so the merger is:

$$out_{N \parallel_{\mathsf{T}} M} = n, n', m, m', m'', \ldots$$

The records between two consecutive delimiters are referred to in the remainder of this chapter as a *substream*. Substreams can, as illustrated above, contain zero or more records. We represent a substream by a list of records:

$$\texttt{type}\ \mathfrak{s} \triangleq [\mathcal{R}]$$

A stream is now represented by a list of substreams:

$$\texttt{type}\ \mathfrak{S} \triangleq [\mathfrak{s}]$$

Because the intuition of functions on a stream is that of applying a function on consecutively arriving records (without any other structural information), we introduce a few higher order functions for streams. We want to allow functions that have some sort of state (as we will see for functions that take an oracle, or for

sychrocells) and that are multiplicitous. Thus, a function of a state and a record is to be 'folded' over a stream. The result of the function is the modified state and the (multiplicitous) response. Since we want to model infinite streams, the folding function does not require an externally observable state. We define `foldS` as a *fold* over streams for non-multiplicitous functions and, similarly, we define `foldSM` for multiplicitous functions. The types for these *fold*-functions are kept general, so that we can also use it for other stream-like data structures (if not, `a` and `b` in the following description would both be $\mathcal{R}$).

```
foldS :: (s → a → (s,b)) → s → [[a]] → [[b]]
foldS _ _    []  ≙ []
foldS f s (σ:σs) ≙ τ : foldS f s' σs
  where
  (s',τ) ≙ foldss f s σ

foldss :: (s → a → (s, b)) → s → [a] → (s,[b])
foldss f s   []  ≙ (s  ,  [])
foldss f s (r:σ) ≙ (s'',r':τ)
  where
  (s' ,r') ≙ f s r
  (s'',τ ) ≙ foldss f s' σ

foldSM :: (s → a → (s,[b])) → s → [[a]] → [[b]]
foldSM f s ≙ map concat ∘ foldS f s
```

Similarly, mapping stateless functions on streams can be defined with the functions `mapS` and `mapSM`. Both are specified as a special `foldS` (resp. `foldSM`) case, where an unchanging unit-type state is added to the function. The effect is, that the function argument of `mapS` (and `mapSM`) is applied to all records in a stream with no data passing between separate function applications. The same holds for the stream equivalent of `filter`: `filterS`.

```
mapS :: (a → b) → [[a]] → [[b]]
mapS f ≙ foldS (λ () r → ((),f r)) ()

mapSM :: (a → [b]) → [[a]] → [[b]]
mapSM f ≙ foldSM (λ () r → ((),f r)) ()

filterS :: (a → Bool) → [[a]] → [[a]]
filterS p ≙ foldSM (λ () r → ((),if p r then [r] else [])) ()
```

6.4.3 Making everything deterministic: oracles

The idea behind the presented formulation of the semantics of SNet is that non-determinism can be expressed as a deterministic choice made outside of the network. Since there are many places in a network where such choices apply, we annotate every choice with a *network index*, to indicate to which part of the network the choice

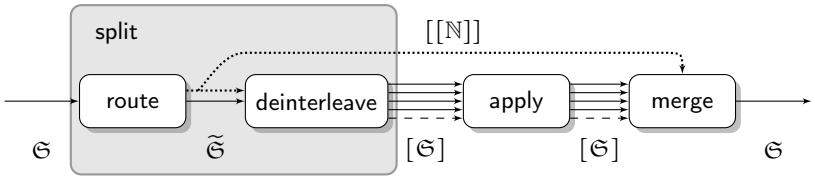


Figure 6.2 – Split-merge pattern

applies. The structural definition of network indices is discussed in section B.3. The oracle for a network is represented by that network's index and the list of annotated choices, where choices are represented by natural numbers.

```
type Ω ≙ (Ψ, [(Ψ,ℕ)])
```

When non-deterministic choices are made by a network, the network queries its oracle and only looks at the choices that apply to it. Thus, we define:

```
query :: Ω → [ℕ]
query (ψ,os) ≙ map snd (filter ((ψ =) ∘ fst) os)
```

The oracle should be thought of simply as a list of choices. To emphasize this perspective, we introduce a restriction operator (↾). The restriction operator takes as its operands an oracle and a network index element (see section B.3). From the network indicated by the network index in the oracle, the oracle is restricted to those networks that can be reached from the network indexed by the network index append with the network index element. Since network indices are partially ordered and networks that can be reached from a point indicated by a network index have a larger network index, we can determine reachability using the infimum (`inf`) of network indices (as defined in section B.3).

```
(↾) :: Ω → NetIdxEl → Ω
(ψ,cs)↾i ≙ (ψ',cs')
  where
  ψ'  ≙ ψ ⊕ i
  cs' ≙ filter ((ψ' =) ∘ inf ψ' ∘ fst) cs
```

6.4.4 A common pattern for combinators: split-merge

There is a basic pattern to the splitting and merging of streams. Because all networks have one input stream and one output stream, streams are split to feed records into different networks. These networks are referred to as branches in the context of splitting and merging. When we represent a branch as a function from stream to stream, we can say that we apply the branch to its stream. The 'split-merge pattern'

is illustrated in figure 6.2. What is illustrated in the figure is not an SNET network, but a visualization of how the functions of the split-merge pattern relate.

Incoming streams must be routed, resulting in a 'routed stream'. In a routed stream all records are annotated with the number of the branch they must flow into:

```
type 𝔖̃ ≙ [[(ℕ,ℛ)]]
```

The routed stream is then deinterleaved according to the routing annotation. What is referred to as a 'split' or 'splitter' in SNET terminology is the combination of the route and deinterleave functions in the figure. Splitting results in a list of *infinitely many* streams (required to describe the inspection combinator, as discussed in section 6.5). The branches are applied to their respective streams. If there are a finite number of branches, the result is a finite list of branch output streams, i.e. when there is no corresponding branch for a stream coming out of the split, that stream is discarded. Finally, the streams are merged back into a single stream. Choices made with regards to routing are important, so the merger gets these choices as an input. The choices are extracted from the routed stream by:

```
routes :: 𝔖̃ → [[ℕ]]
routes ≙ mapS fst
```

The (non-)determinism of the split-merge pattern is two-fold. On the one hand, splitting is non-deterministic when there are multiple branches into which a record can flow and, on the other hand, the merge-order maybe non-deterministic. Splitting non-determinism is decided in routing, i.e. it is clear for every record in the routed stream into which branch it should flow. Merging non-determinism has implications for both the splitter and the merger. For example, in a deterministic parallel composition, every single record flowing into a branch is its own substream (recall the example given in section 6.4.2). In the alternative case, the non-deterministic parallel composition, all branches receive all substreams that arrived at the splitter, but per substream only those records that are routed there.

We illustrate non-deterministic split-merge with a variation of the example in section 6.4.2. We assume that the parallel composition of networks N and M is non-deterministic. Also, we add to the input stream a delimiter, a record d and another delimiter. The responses of network N and M are the same as above (with the addition of the response to d: the single record n'').

$$\begin{aligned} in_{N \parallel_{\mathsf{F}} M} &= a, b, c \sqsquare d \sqsquare \dots \\ in_N &= a, c \sqsquare d \sqsquare \dots \\ in_M &= b \sqsquare \sqsquare \dots \\ out_N &= n, n' \sqsquare n'' \sqsquare \dots \\ out_M &= m, m', m'' \sqsquare \sqsquare \dots \end{aligned}$$

The non-deterministic split-merge does *not* introduce delimiters between every record, because the merger may arbitrarily interleave out_N and out_M. However,

delimiters in $in_{N \parallel_F M}$ must be honoured by the merger. The output stream of the merger, therefore, is

$$out_{N \parallel_F M} = \frac{n, n'}{m, m', m''} \square\, n'' \square \ldots$$

where the notation before the first delimiter indicates an arbitrary interleaving of n, n' and m, m', m''.

We first give the types of the functions implementing the split-merge pattern, starting with the function for the entire pattern `splitMerge`.

```
splitMerge :: Bool → Router → [Branch] → Ω → 𝔖 → 𝔖
```

It requires a router function that, as described above, may make non-deterministic choices. Therefore, the router function takes as an extra argument an oracle. Since branches may have their own internal non-determinism, they also require an oracle besides their branch number and their input stream. With its branch number, a branch can restrict its own oracle.

```
type Router ≙      Ω → 𝔖 → 𝔖̃
type Branch ≙ ℕ → Ω → 𝔖 → 𝔖
```

Both the deinterleaver and the merger need to be informed on whether this is a deterministic split-merge pattern. Only the merger requires an oracle.

```
deinterleaveS :: Bool →                  𝔖̃  → [𝔖]
mergeS        :: Bool → [[ℕ]] → Ω → [𝔖] →  𝔖
```

The router and merger are applied to an oracle that is restricted to their respective network index elements (`Split` and `Merge` as described in section B.3). As described above, the branches receive as an argument their branch number, an oracle and their respective stream.

```
splitMerge δ routeS branches ω s ≙ let
    routed ≙ routeS (ω↾Split) s
    splits ≙ deinterleaveS δ routed
    applyS ≙ λb i s → b i ω s
    result ≙ zipWith3 applyS branches [0..] splits
  in mergeS δ (routes routed) (ω↾Merge) result
```

Deinterleaving

As discussed above, deinterleaving depends on whether or not the split-merge is deterministic. If the split-merge is non-deterministic, deinterleaving introduces no new substream divisions, i.e. delimiters. Instead, every branch receives every substream, but filters out the records routed to other branches. If the split-merge is deterministic, every record is mapped to a singleton substream and branches are fed only the substreams that contain a record that is routed to that branch.

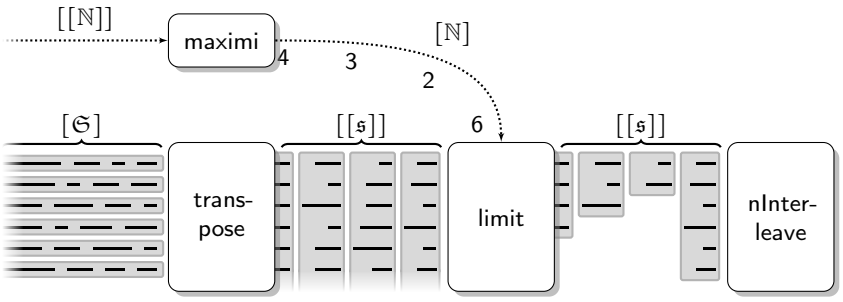


FIGURE 6.3 – Non-deterministic merger

```
deinterleaveS δ rs ≙ [ delims (filterS ((=i).fst) rs) | i ← [0..] ]
  where
  delims :: 𝔖̃ → 𝔖
  delims | ¬δ ≙          mapS          snd
         |  δ ≙ concat ∘ mapS ((:[]) ∘ snd)
```

Merging

Again, merging comes in two variants: deterministic and non-deterministic. Both are described below, but first we introduce a helper function that is used by both variants of merger: The function `applyAt` applies a function to precisely one element of a list, indicated by the element's index in that list.

```
applyAt :: (a → a) → ℕ → [a] → [a]
applyAt f 0 (x:xs) ≙ f x : xs
applyAt f n (x:xs) ≙ x : applyAt f (n-1) xs
```

Merging the split streams deterministically requires only an interleaving of the substreams produced by the branches, according to the routing information. By `foldSM`ing `dInterleave` over the routing choices with the streams to be merged as its state, the result is the merged stream.

```
dInterleave :: [𝔖] → ℕ → ([𝔖],𝔰)
dInterleave σss c ≙ (applyAt tail c σss, head (σss!!c))
```

A non-deterministic merger merges all incoming substreams into a single substream. The structure of non-deterministic mergers is illustrated in figure 6.3. The (potentially infinite) list of incoming streams is transposed, resulting in (potentially infinite) lists of substreams. The decisions made in the router are used to limit the length of these substreams: For every substream, the maximum branch number indicates the highest numbered stream that contains a non-empty substream. After

limiting, the list of finite lists of substreams are interleaved per list into a single substream.

Transposition and limiting are both deterministic.


```
transpose :: [𝔖] → [[Maybe 𝔰]]
transpose mx ≙ map head' mx : transpose (map tail mx)
  where
  head'    []  ≙ Nothing
  head' (x:xs) ≙ Just x

limit :: [[ℕ]] → [[Maybe 𝔰]] → [[𝔰]]
limit choices ≙ map (map fromJust) ∘ takeWhile (all isJust) ∘
  zipWith take (map maximum choices)
```

`nInterleave` is non-deterministic. It requires as an argument the list of choices dictated by the merger's oracle. The input is a list of lists of substreams. Every list of substreams can have a different length (determined by limiting). When all substreams in the 'current' list (`subs`) are empty, the end of the resulting substream is reached and `nInterleave` continues with the next substream. If ever the oracle decision points to an empty substream, the result is the end of the entire stream. This is required behaviour to attain the property of prefix monotonicity, discussed in section 6.6.

```
nInterleave :: [ℕ] → [[𝔰]] → 𝔖
nInterleave  _ [] ≙ []
nInterleave cs (subs:subss) | all null subs ≙ nInterleave cs subss
nInterleave (c:cs) (subs:subss)
    | null sub  ≙ [[]]
    | otherwise ≙ applyAt (head sub :) 0 subss'
  where
  sub    ≙ subs!!c
  subss' ≙ nInterleave cs (applyAt tail c subs : subss)
```

Now both types of merger can be combined in `mergeS`. At the top-level, the choice is made for deterministic or non-deterministic merger. If the merger is deterministic, as discussed above, we `foldS` the `dInterleave` function over the router choices. If the merger is non-deterministic, first, the incoming list of streams is transposed and limited (`subss`). The oracle is queried to provide all the choices for the `nInterleave` function.

```
mergeS δ choices ω σss |  δ ≙ foldSM dInterleave σss choices
                       | ¬ δ ≙ nInterleave (query ω) subss
  where
  subss ≙ limit choices (transpose σss)
```

6.4.5 SYNCHRONISATION

Whereas merging combines streams, synchronisation combines records. Synchronisation occurs in synchrocells. Synchrocells carry with them a list of patterns to match (`pats`). There is flow inheritance for synchrocells: The record to match the first pattern in `pats` is flow inherited over the synchrocell when the synchrocell produces a result. For records that only match one or more of the other patterns in `pats`, the matching fields are stored and the unmatched fields are discarded. First, we describe the state of synchrocells.

Given a list of patterns, the state of a synchrocell can be initialised. Every pattern is translated into a matching function, which takes a record and tries to match it to the pattern. If a match succeeds, the result of matching is a record type. The record is split into an inheritance structure, containing a `through` and an `around` part. The state of a synchrocell is a list of either matching functions, that have not (yet) been matched, or inheritance structures resulting from successful earlier matches.

```
type SyncState ≙ [Either (ℛ → Maybe ℑ) ℑ]

syncInit :: [𝔓 (Maybe 𝔑)] → SyncState
syncInit pats ≙ map (Left ∘ λp r → fmap (≺ r) (eval r p)) pats
```

Next, we describe the transformation of a synchrocell's state (`syncS`). There are three conditions for the state to which `syncS` shows different behaviour. When the state is an empty list, this indicates the synchrocell has already produced a result in response to a record to which the synchrocell was applied before. When the state is a non-empty list, there are two possiblities: Either the incoming record matches all remaining patterns, leaving nothing left `todo`, but to produce a result *or* there are unmatched patterns after the incoming record has been absorbed and no result is produced. If a `result` is produced, it consists of the combination of all (`through` parts of) records stored earlier, with the fields that flow inherit from the record that matched the first pattern in `pats`.

```
syncS :: SyncState blv → ℛ → (SyncState blv,[ℛ])
syncS []    r ≙ ([],[r])
syncS state r | null todo ≙ ([],[result])
              | otherwise ≙ (state',[])
  where
  (todo,done) ≙ partitionEithers state'
  state' ≙ map (either (λp → maybe (Left p) Right (p r)) Right) state
  result ≙ head done ≻ foldl1 combine (map through done)
```

6.5 Semantics

Using the data structures and utilities described above, the semantics of a network can be written as a function of that network and its index. In other words, the

function $[\![.]\!]$ describes unambiguously the behaviour of any network. The behaviour is again a function that takes an oracle and a stream and produces a stream.

$$[\![.]\!] :: \mathcal{N} \to \Omega \to \mathfrak{S} \to \mathfrak{S}$$

In the following sections, we give a definition for $[\![.]\!]$ for every primitive network and every network combinator.

6.5.1 PRIMITIVE NETWORKS

In this section, we define the semantics of primitive networks and use these definitions for the semantics of network combinators in the following sections. We start with boxes. Boxes have no non-determinism, so the oracle is not needed for any box. However, flow inheritance occurs on boxes. Records flowing into a box are split according to the box type $\mathfrak{b}$. The result is an inheritance structure (`inh`). This inheritance structure contains two parts: `through` and `around`. The `through` part is fed into the box and the `around` part is merged with all records coming out of the box. To feed the `through` part of the record (which is itself a record), it must be converted into box input (`toBox`). The box may be multiplicitous, i.e. the box' output can consist of multiple records. All of the box' output must be converted back to records (`fromBox`) and the `around` part is flow inherited over the box (`inh` $\succ$).

$$[\![\textbf{box}(\mathfrak{b}, f)]\!] \triangleq \lambda\_\ \texttt{s} \to \texttt{mapSM applyBox s}$$
$$\texttt{where}$$
$$\texttt{applyBox r} \triangleq \texttt{let inh} \triangleq \mathfrak{b} \prec \texttt{r in}$$
$$(\texttt{map}\ ((\texttt{inh} \succ) \circ \texttt{fromBox}\ \mathfrak{b}) \circ f \circ \texttt{toBox}\ \mathfrak{b} \circ \texttt{through})\ \texttt{inh}$$

The semantics of filters is (intuitively) the same as that of boxes, besides the conversions of records to box input and box output to records. Filters are defined by a list of actions $\mathcal{A}$ that can be evaluated for every incoming record.

$$[\![\textbf{filter}(\mathfrak{f}, \mathcal{A})]\!] \triangleq \lambda\_\ \texttt{s} \to \texttt{mapSM applyFilter s}$$
$$\texttt{where}$$
$$\texttt{applyFilter r} \triangleq \texttt{let inh} \triangleq \mathfrak{f} \prec \texttt{r in map}\ (\texttt{inh} \succ)\ (\texttt{eval}\ (\texttt{through inh})\ \mathcal{A})$$

Synchrocells are primitive networks with a state. The initial state comes from the synchrocell's description, i.e. patterns given in $\mathcal{P}$. The behaviour of a sychrocell is defined per record, given the propagation of a state. This behaviour is implemented in the function `syncS` described above.

$$[\![\textbf{sync}(\mathcal{P})]\!] \triangleq \lambda\_\ \texttt{s} \to \texttt{foldSM syncS}\ (\texttt{syncInit}\ \mathcal{P})\ \texttt{s}$$

6.5.2 SEQUENTIAL COMPOSITION

Sequential composition of networks is akin to function composition.

```
⟦ ••_{N←L} N ⟧ ≙ λω s → let
    L' ≙ zipWith (λ N i → ⟦N⟧ (ω↾Seq i)) L [0..]
  in (foldr1 ( ∘ ) L') s
```

6.5.3 PARALLEL COMPOSITION

Parallel composition adheres to the split-merge pattern (see section 6.4.4). Branch numbers are encoded in the index as alternatives (`Alt`). Routing is based on the strongest matching network type of the branches and is implemented in routing function `router`. When records match multiple branches' network types equally well, the oracle is queried to choose from those branches with the strongest matching network types.

```
⟦ ||_{δ} N ⟧_{N←L} ≙ splitMerge δ (router ∘ query) (map branch L)
  where
    types ≙ map netType L
    branch N i ω ≙ ⟦N⟧ (ω↾Alt i)
    router ≙ foldS route
      where
      route :: [ℕ] → ℛ → ([ℕ], (ℕ, ℛ))
      route cs r ≙ (cs',(options!!c,r))
        where
        matches ≙ map (match r) types
        hiscore ≙ maximum matches
        alts    ≙ zip matches [0..]
        options ≙ [ c | (m,c) ← alts, m = hiscore ]
        (c:cs') | null (tail options) ≙ (0:cs)
                | otherwise           ≙ cs
```

6.5.4 SERIAL REPLICATION

The serial replication has terminator patterns (γ). If any of those patterns match the incoming record, that record bypasses all following instance of the replicated network. If no patterns match the incoming record, the record is passed through another instance of the replication. This means the serial replication fits the split-merge pattern (see section 6.4.4). The router (`stop`) decides where to route a record, based on whether or not it matches the terminator patterns. The intuition for the serial replication is that of recursion or iteration. However, because we want to use Haskell's denotational semantics, we may not make the rewriting of SNet programs an infinite process: Only *finite* Haskell programs have well-defined semantics. Therefore, the semantics may not be recursive in itself. Instead, we define `net i` to represent the i^{th} unrolling of the serial replication's body. The function `net` is given a branch number as its second argument, because of the definition of `splitMerge`.

This branch number serves no purpose in disambiguating instances (which is what argument `i` is for). Therefore, the branch number is ignored.

```
⟦N*_{δ,γ}⟧ ≙ net 0 0
  where
  net i _ ω ≙ splitMerge δ xt [net (i+1), λ_ _ → id] ω ∘ ⟦N⟧ (ω↾Iter i)
  xt _ ≙ mapS (λr → (if done r then 1 else 0,r))
    where
    done r ≙ any (maybe False (const True)) (map (eval r) γ)
```


6.5.5 inspection composition

The inspection combinator also adheres to the split-merge pattern (see section 6.4.4). The splitter, in this case, is fully deterministic: The value of tag `sel` is used as the branch number to which to route. Because the value of tag `sel` can be arbitrarily large, there may be arbitrarily many branches. This is the reason why the split-merge pattern assumes infinitely many branches and why non-deterministic mergers must allow a different number of branches to merge for every substream.

```
⟦N !δ t⟧≙ splitMerge δ router (repeat branch)
  where
  router _ ≙ mapS (λr → (route r, r))
  route  r ≙ let z ≙ eval r (E_L sel) in 2 * abs z + div (signum z - 1) 2
  branch i ω ≙ ⟦N⟧ (ω↾Alt i)
```

The function $[\![.]\!]$ is a *total function* from all primitive networks and network combinators to Haskell functions from an oracle and a stream to a stream. These streams consist of substreams (as described in section 6.4.2). However, the intuitive interpretation of a stream is a list of records. Therefore, we want an entry-point into the semantics that adds this structural information (substreams) to the input and removes it from the output. Substreams arise from deterministic split-merge instances. Thus, for a stream from the 'outside world' we can say that the entire stream is precisely one substream. Moreover, under the restriction that all networks must preserve delimiters, the result is a stream with precisely one substream. This leads to:

```
⟦.⟧‾ :: 𝒩 → [(Ψ,ℕ)] → [ℛ] → [ℛ]
⟦N⟧‾ ω ≙ head ∘ ⟦N⟧ ([],ω) ∘ (:[])
```

6.6 Prefix monotonicity

With the semantics presented in the previous section, we can prove that every SNet network is *prefix monotonic*. We first define two relations that express prefix monotonicity.

Definition.(List prefix) The list prefix relation, denoted by $xs \sqsubseteq ys$ for prefix xs of ys, is defined as

$$\begin{array}{l} x : xs \sqsubseteq y : ys \quad \textit{iff} \quad x = y \wedge xs \sqsubseteq ys \\ [\,] \sqsubseteq ys \end{array}$$

Definition.(Nested-list prefix) The nested-list prefix relation, denoted by $xss \trianglelefteq yss$ for prefix xss of yss, is defined as

$$\begin{array}{l} xs : xss \trianglelefteq ys : yss \quad \textit{iff} \quad (xs = ys \wedge xss \trianglelefteq yss) \vee (xs \sqsubseteq ys \wedge xss = [\,]) \\ [\,] \trianglelefteq yss \end{array}$$

With these definitions, we can express what we mean by saying that every SNet network is prefix monotonic: Let N denote an arbitrary SNet network, rs a list of records and s the *state* of the network, then

$$rs' \sqsubseteq rs \quad \Rightarrow \quad \forall \omega \left(\overline{[\![N]\!]}_s \; \omega \; rs' \sqsubseteq \overline{[\![N]\!]}_s \; \omega \; rs \right)$$

The state in the denotational semantics given in the previous section is not externally observable, but the intuition is that s is the collection of all states at all hierarchical levels in N.

The overall structure of the proof is by two-level induction. On the top-level we proceed by induction on the structure of the network N. Then we proceed by induction on the length of the prefix rs' of rs. The basic case is $rs' = [\,]$, i.e. we prove that

$$\overline{[\![N]\!]}_s \; \omega \; [\,] \sqsubseteq \overline{[\![N]\!]} \; \omega \; rs$$

For the induction case we may assume the induction hypothesis, i.e. *for all s* we have that

$$\overline{[\![N]\!]}_s \; \omega \; rs' \sqsubseteq \overline{[\![N]\!]}_s \; \omega \; rs$$

and have to prove *for all s*:

$$\overline{[\![N]\!]}_s \; \omega \; (r : rs') \sqsubseteq \overline{[\![N]\!]}_s \; \omega \; (r : rs)$$

The reason for emphasising "for all s" above, is that the effect of r may be that the state s in the network changes. Thus, in order to apply the induction hypothesis, it should hold for any state that may be the result of processing r by N. Clearly, it should be possible that r is executed in any state s as well.

It follows from the definitions of $\overline{[\![.]\!]}$ and $\sqsubseteq$, that

$$\overline{[\![N]\!]}_s \; \omega \; rs' \sqsubseteq \overline{[\![N]\!]}_s \; \omega \; rs \quad \Leftrightarrow \quad [\![N]\!]_s \; \omega \; [rs'] \trianglelefteq [\![N]\!]_s \; \omega \; [rs]$$

We use this property for the induction over the structure of N. Also, in the following, σ denotes a substream and σs denotes a stream. Thus $\sigma : \sigma s$ denotes a stream with

at least one substream. Using the right-hand-side of the expression above, however, means that there are two induction case: In the first, we add a substream to the stream, whereas in the second, we add a record to the first substream. This means we have three cases to prove for the induction case:

$$\begin{array}{lccl} \mathcal{C}_1: & \varpi s' \trianglelefteq \varpi s & \Rightarrow & \sigma : \varpi s' \trianglelefteq \sigma : \varpi s \\ \mathcal{C}_2: & \sigma' : \varpi s' \trianglelefteq \sigma : \varpi s & \Rightarrow & \sigma' = \sigma \wedge \varpi s' \trianglelefteq \varpi s \\ \mathcal{C}_3: & \sigma' : \varpi s' \trianglelefteq \sigma : \varpi s & \Rightarrow & \sigma' \sqsubseteq \sigma \wedge \varpi s' = [\,] \end{array}$$

6.6.1 Proof for SNet networks

Because the semantics of SNet networks is defined in terms of a few utility functions, we first prove that these functions are prefix monotonic.

Lemma 1 (*foldss*) $\quad \sigma' \sqsubseteq \sigma \;\Rightarrow\; \forall f, s\,(snd\,(foldss\ f\ s\ \sigma') \sqsubseteq snd\,(foldss\ f\ s\ \sigma))$

Proof: By induction with base case $\sigma' = [\,]$, trivial. For the induction case, let

$$(s', r') = f\ s\ r$$

and let

$$\begin{aligned} (s_1, \sigma_1) &= foldss\ f\ s\ \sigma' \\ (s_2, \sigma_2) &= foldss\ f\ s\ \sigma \end{aligned}$$

Thus

$$\begin{aligned} foldss\ f\ s\ (r : \sigma') &= (s_1, r' : \sigma_1) \\ foldss\ f\ s\ (r : \sigma) &= (s_2, r' : \sigma_2) \end{aligned}$$

Induction hypothesis:

$$\sigma_1 \sqsubseteq \sigma_2$$

Hence, taking the *snd* on both sides gives

$$r' : \sigma_1 \sqsubseteq r' : \sigma_2$$

□

Lemma 2 (*foldS*) $\quad \varpi s' \trianglelefteq \varpi s \;\Rightarrow\; \forall f, s\,(foldS\ f\ s\ \varpi s' \trianglelefteq foldS\ f\ s\ \varpi s)$

Proof: By induction with base case $\varpi s' = [\,]$, trivial. For the induction case $\mathcal{C}_1$, i.e.

$$\begin{aligned} &\forall f, s(foldS\ f\ s\ \varpi s' \trianglelefteq foldS\ f\ s\ \varpi s) \\ &\qquad\Rightarrow\; \forall f, s(foldS\ f\ s\ (\sigma : \varpi s') \trianglelefteq foldS\ f\ s\ (\sigma : \varpi s)) \end{aligned}$$

trivial. For induction case $\mathcal{C}_2$, let

$$(s', \tau') = foldss\ f\ s\ (r : \tau)$$

and let

$$\tau s' = foldS\ f\ s'\ \sigma s'$$
$$\tau s = foldS\ f\ s'\ \sigma s$$

then

$$foldS\ f\ s\ ((r : \tau) : \sigma s') = \tau : \tau s'$$
$$foldS\ f\ s\ ((r : \tau) : \sigma s)\ = \tau : \tau s$$

Proof follows from the induction hypothesis:

$$\tau s' \sqsubseteq \tau s$$

For induction case $\mathcal{C}_3$ let

$$(s_1, \tau_1) = foldss\ f\ s\ (r : \sigma')$$
$$(s_2, \tau_2) = foldss\ f\ s\ (r : \sigma)$$

hence (by lemma 1)

$$\tau_1 \sqsubseteq \tau_2$$

Thus

$$\tau_1 : [\,] \trianglelefteq \tau_2 : \tau s$$

□

Lemma 3 (*foldSM*) $\quad \sigma s' \trianglelefteq \sigma s \ \Rightarrow\ \forall f, s\,(foldSM\ f\ s\ \sigma s' \trianglelefteq foldSM\ f\ s\ \sigma s)$

Proof: With lemma 2, it is easy to see that

$$\sigma s' \trianglelefteq \sigma s \ \Rightarrow\ \forall f, s\,(map\ concat\ (foldS\ f\ s\ \sigma s') \trianglelefteq map\ concat\ (foldS\ f\ s\ \sigma s))$$

□

Lemma 4 $\quad \sigma s' \trianglelefteq \sigma s \ \Rightarrow\ \forall f, s$

$$mapS\ f\ s\ \sigma s' \trianglelefteq mapS\ f\ s\ \sigma s$$
$$mapSM\ f\ s\ \sigma s' \trianglelefteq mapSM\ f\ s\ \sigma s$$
$$filterS\ f\ s\ \sigma s' \trianglelefteq filterS\ f\ s\ \sigma s$$

Proof: Immediate from lemmas 2 and 3 □

Lemma 5 (*splitMerge*) $\quad \sigma s' \trianglelefteq \sigma s \ \Rightarrow\ \forall f, s$

$$splitMerge\ F\ R\ bs\ \omega\ \sigma s' \trianglelefteq splitMerge\ F\ R\ bs\ \omega\ \sigma s$$

Proof: Examine the non-deterministic case, i.e. $\delta = \mathsf{F}$. The incoming stream is split, such that for every substream σ, there is a (possibly empty) substream σ_i in the

input stream of branch $b_i \in bs$. Every record in σ occurs in precisely on σ_i. From the definition, it follows that

$$\begin{aligned} &splitMerge\ F\ R\ bs\ \omega\ \sigma s' \\ &\quad = mergeS\ F\ css\ \omega\ [b_i\ \sigma s'_i \mid i \leftarrow [0, \ldots]] \\ &\quad = mergeS\ F\ css\ \omega\ [\tau s'_i \mid i \leftarrow [0, \ldots]] \end{aligned}$$

We go on to show that

$$\begin{aligned} &\forall i (\tau s'_i \trianglelefteq \tau s_i) \ \Rightarrow\ \forall css, \omega \\ &\quad mergeS\ F\ css\ \omega\ [\tau s'_i \mid i \leftarrow [0, \ldots]] \trianglelefteq mergeS\ F\ css\ \omega\ [\tau s_i \mid i \leftarrow [0, \ldots]] \end{aligned}$$

For induction case $\mathcal{C}_1$:

$$\begin{aligned} &mergeS\ F\ cs : css\ \omega\ [\tau_i : \tau s'_i \mid i \leftarrow [0, \ldots]] \\ &\quad = nInterleave\ (query\ \omega) \\ &\qquad\qquad (limit\ (cs : css)\ (transpose\ [\tau_i : \tau s'_i \mid i \leftarrow [0, \ldots]])) \\ &\quad = nInterleave\ (query\ \omega)\ (\overbrace{[\tau_i \mid i \leftarrow [0, \ldots, maximum\ cs]]}^{\rho} \\ &\qquad\qquad : limit\ css\ (transpose\ [\tau s'_i \mid i \leftarrow [0, \ldots]])) \end{aligned}$$

If *nInterleave* fails to interleave the full substream ρ, the result is $[\rho']$. On the other hand, if *nInterleave* succeeds, there exists an oracle ω', such that the result is

$$nInterleave\ (query\ \omega)\ \rho : mergeS\ F\ css\ \omega'\ [\tau s'_i \mid i \leftarrow [0, \ldots]]$$

It follows that

$$\begin{aligned} &\forall i, css, \omega (b_i\ \sigma s'_i \trianglelefteq b_i\ \sigma s_i\ \Rightarrow \\ &\quad mergeS\ F\ css\ \omega\ [b_i\ \sigma s'_i \mid i \leftarrow [0, \ldots]] \trianglelefteq mergeS\ F\ css\ \omega\ [b_i\ \sigma s_i \mid i \leftarrow [0, \ldots]]) \end{aligned}$$

The proof of the other induction cases is tedious, but straightforward and similar to the previous proofs. The same holds for the deterministic case of *splitMerge*. □

Lemma 6 (*function composition*)

$$\begin{aligned} &\sigma s' \trianglelefteq \sigma s\ \rightarrow\ f(\sigma s') \trianglelefteq f(\sigma s) \wedge g(\sigma s') \trianglelefteq g(\sigma s) \\ \Rightarrow\ &\sigma s' \trianglelefteq \sigma s\ \rightarrow\ (f \circ g)(\sigma s') \trianglelefteq (f \circ g)(\sigma s) \end{aligned}$$

Proof: Let

$$\begin{aligned} \tau s &= g(\sigma s) \\ \tau s' &= g(\sigma s') \end{aligned}$$

it follows that

$$\tau s' \trianglelefteq \tau s$$

and

$$f(\tau s') \trianglelefteq f(\tau s)$$

□

Theorem.(*Prefix monotonicity of N*)

$$\forall \omega \left(\sigma s' \trianglelefteq \sigma s \;\Rightarrow\; [\![N]\!]_s \;\omega\; \sigma s' \trianglelefteq [\![N]\!]_s \;\omega\; \sigma s' \right)$$

Proof: By induction over the structure of N. For every clause of $[\![.]\!]$:

» $[\![\mathbf{box}(\mathfrak{b}, f)]\!]$, $[\![\mathbf{filter}(\mathfrak{f}, \mathcal{A})]\!]$, $[\![\mathbf{sync}(\mathcal{P})]\!]$: Immediate from lemmas 3 and 4.

» $\left[\!\!\left[\overset{\bullet\bullet}{N \leftarrow L} N \right]\!\!\right]$: Immediate from the induction hypothesis and lemma 6.

» $\left[\!\!\left[\overset{\|_\delta}{N \leftarrow L} N \right]\!\!\right]$: Immediate from the induction hypothesis and lemma 5.

» $[\![N^\star_{\delta,\gamma}]\!]$: Immediate from the induction hypothesis and lemmas 5 and 6.

» $[\![N \;!_\delta\; t]\!]$: Immediate from the induction hypothesis and lemmas 5.

□

6.7 Conclusion

In this chapter, we have given a denotational semantics of the language SNet by giving a translation from SNet to the pure, lazy, functional programming language Haskell. In this translation, non-determinism in SNet has been translated to choices represented by an oracle. Given an oracle, the behaviour of a translated SNet network is purely functional, i.e. fully deterministic. Using the presented denotational semantics, a proof has been provided that, for every oracle, every SNet network is prefix monotonic.

Hydra: an SNet implementation

Abstract – *Two problems are identified in the current execution model of SNet. A new execution model for SNet is developed in this chapter. This new execution model is implemented in Hydra, a combination of a compiler and run-time system. A strong indication is given at the end of this chapter, that Hydra does not introduce non-termination.*

7.1 Motivation

This chapter discusses Hydra, an SNet implementation with a different execution model than the one implemented in the current SNet-compiler, `snetc` [76, 86]. The `snetc` execution model takes an intuitive approach to threading: primitive networks are threads. Every primitive network has an input buffer into which records can be written. When there is a record in a thread's input buffer, that thread can consume it, perform the operations specified by the corresponding primitive network and produce output in the input buffers of the threads handling primitive networks to which the records flow next. Thus, the perspective of an SNet network is that of a network of communicating processes. Typically, `snetc` compiles for shared memory architectures, but more recent work [36, 38] describes distributed memory approaches.

Parts of an earlier revision of this chapter have been published in [PhH:9].

Hydra seeks to solve two problems with the `snetc` execution model. The first problem is the resource management. Consider, as an example, the network N .. M, where N and M are both boxes. Two threads are instantiated for this network, t_N and t_M, respectively. If N imposes a heavy computational load, while M is a very light-weight computation, t_N will be constantly busy, while t_M is idle most of the time. Utilization can be improved at run-time, by observing the input buffers. When a resource manager finds that t_N's buffer is full most of the time, while that of t_M is (close to) empty, it is clear that t_N is a bottleneck. The network can be dynamically reconfigured to $N \parallel_{\mathsf{T}} N$ .. M, so that a new thread t'_N can be started and the throughput of N improves. This solution does not scale well to distributed memory architectures, because it requires resource management to observe input buffers across a distributed system. Although a strategy may be used to promote locality of communicating threads, for networks in which loads are data dependent and highly variable, fragmentation is inevitable.

Another instance of the same problem is that of reclaiming resources. Consider the network $(\mathsf{sync}\,(\{A\},\{B\})$ .. $N \parallel_{\mathsf{T}} Id)^{\star}_{\{X\},\gamma}$, where N is a network that takes records of type $\{A,B\}$ and produces records of type $\{X\}$. The synchrocell combines records of type $\{A\}$ with records of type $\{B\}$ to form records of type $\{A,B\}$. After the first, all other records of type $\{A\}$ bypass network N (by flowing through the identity network *Id*), do not match the exit pattern $\{X\}$ and flow into the next instance of the serial replication. The same holds for records of type $\{B\}$. This means only the first records of either type uses the first instance of N, while all following records bypass it and flow into consecutive instances. As this network is specified to have two input variants, $\{A\}$ and $\{B\}$, also records of the subtype $\{A,B\}$ are allowed as input. Therefore, it can not be guaranteed that, after the synchrocell syncs, the first instance of N gets no further input. The thread handling this instance can thus not terminate and its resources can not be reclaimed. A few specific cases of this type of problem can be recognized and optimized out by `snetc`, but in the general case (for arbitrary N), the problem remains.

The second problem is a distribution overhead problem. When a record is sent from one memory domain to another, it must be serialized for communication. Serialization in many homogeneous environments is a straightforward copy of memory. However, in cases where serialization requires processing, it can not be delegated to DMA or similar hardware solutions. For SNet networks with light-weight boxes on large amounts of data, the overhead from serialization can become a bottleneck.

7.2 Approach

The approach of Hydra's execution model is to take an orthogonal perspective on threading: Instead of having a thread for every network, taking records as input and producing records for consecutive networks, a thread is created for every incoming

record. This thread carries its record through the SNet network and applies boxes to its record along the way. Using this perspective, inter-thread communication no longer consists of records, but of synchronization messages. In other words, Hydra's execution model brings networks to records, instead of bringing records to networks.

In the first example network above, *N* .. *M*, records coming into the network no longer need to wait for records that arrived earlier. Every time a record flows into the network, a thread is started that applies first *N* and then *M* to its record. This means records may arrive at the network's output out-of-order, so at the output, the order needs to be re-established. At this point, threads communicate to determine which thread should be the first to produce its record on the output. A similar reordering problem occurs inside a network at synchrocells. However, this reordering problem is a *local* reordering, in the sense that it does not need to establish the order of *all* records, but per synchrocell only the records that flow through that synchrocell. Because reordering at the output *does* require establishing the order of all records, it is considered a *global* reordering.

Both the global reordering at the output and local reordering at synchrocells can be performed with the same mechanism. The input order, i.e. the order in which records arrive at the input of the network, is recorded in a data structure: Records are stored in a *cons-list*[1]. A cons-list is a type of singly-linked list in which every node has a pointer to the data contained at that node (in our case: a record) and a pointer to the next element in the list. The first record to arrive is the first element (or 'head') of the cons-list. Consecutive records are appended to the last element (or 'tail') of the cons-list. The order of the cons-list is never changed. However, when primitive networks are multiplicitous, the causal order is preserved: If the response of a primitive network to a record consists of zero records, the cons-node corresponding to that record is removed and its predecessor is connected to its successor. If the response consists of multiple records, the first is stored at the cons-node of the original record and consecutive records are inserted between that cons-node and its successor.

Hydra's cons-nodes are implemented as *containers*. Besides a pointer to the next cons-node and to the record, containers hold administration variables used for reordering. Every container is managed by a thread. Inter-thread communication is implemented using these administration variables. Only the container's manager thread may read and write the container's variables, with the exception of variables for inter-thread communication: Shared variables in a container may only be *read* by the thread managing that container and may only be *written* by the thread managing the container's predecessor. Such one-way communication can be implemented lock-less and atomic. For the basic execution model of Hydra, this is all inter-thread communication. However, as discussed in section 7.3.3, adding shared variables for synchrocells significantly relaxes the local reordering problem. Besides the shared variables in the containers and the synchrocells, threads share no data. Transactions

[1] The term comes from the programming language Lisp. The term 'cons' is short for 'constructor.'

on these variables and the spawning of new threads are the only interactions between threads.

7.3 Compilation scheme & run-time system

In this section, we construct a compilation scheme and a run-time system, that together implement Hydra's execution model. We start with a compilation scheme and a run-time system for a small subset of the set of SNet combinators and incrementally add combinators, changing both compilation scheme and run-time system where needed. To make clear where the global state is modified—especially where threads are created or destroyed and where threads communicate—the compilation scheme is kept purely functional, whereas the run-time system is written in a procedural way.

The variables in containers are named. The notation c_a represents a variable a in a container c. In procedures,

$$a \leftarrow b$$

denotes assignment of the value b to a *local* variable a, whereas

$$a \twoheadleftarrow b$$

denotes a *global* (or 'persistent') assignment. Procedures within the run-time system are not considered functional: Their arguments can not be curried and they are not first-class citizens, i.e. they can not be passed as values.

Compilation schemes translate networks to functions that take a single container as their argument. Every compilation scheme has an entry-point that introduces handlers for in- and output from the run-time system. The entry-point of a compilation scheme is indicated by a superscripted star. The result of translating a network with the entry-point of a compilation scheme is a self-contained program, i.e. the result does not take any arguments. We first show how stateless sequential networks can be compiled and what the minimum run-time system for such networks consists of.

7.3.1 Stateless sequential networks: output reordering

We start off with compilation scheme $\mathcal{C}_{seq}$, which compiles stateless sequential networks without multiplicity. Boxes in such a network are applied in-order to the incoming records and always produce *one* record. Thus, the order in which the (processed) records are output is the same as the order in which they were input into the network. However, with a thread for every record, the (temporal) order in which records arrive at the output of the network is undefined. This means that the order in which the records arrived at the network's input must be restored at the network's output.

Compilation scheme

First, we look at the compilation of boxes. Since boxes are implemented by some box language which is already compiled, we can say that box (f) indicates a box with a function $f : \mathit{record} \to \mathit{record}$ as its implementation. A compiled box, therefore, is a function that takes a container c and applies f to the record contained in c. The run-time system procedure modifyRecord applies f to the record stored in c_{record} and overwrites c_{record} with the resulting record.

$$\mathcal{C}_{seq}\,[\![\mathsf{box}\,(f)]\!] = \lambda c.\mathsf{modifyRecord}(f, c)$$

Consequently, sequential composition translates to function composition, viz.

$$\mathcal{C}_{seq}\,[\![N \mathbin{..} M]\!] = \mathcal{C}_{seq}\,[\![M]\!] \circ \mathcal{C}_{seq}\,[\![N]\!]$$

What remains is handling the in- and output. For the latter, we introduce a compiler known network out, a procedure handleOutput in the run-time system and the corresponding compilation rule

$$\mathcal{C}_{seq}\,[\![\mathsf{out}]\!] = \lambda c\ .\ \mathsf{handleOutput}(c)$$

For the former, we introduce a procedure handleInput in the run-time system, that takes as an argument the continuation [92, 96] that is applied to every record coming into the run-time system. Herewith, the entry-point of the compilation can be written as

$$\mathcal{C}^{\star}_{seq}\,[\![N]\!] = \mathsf{handleInput}(\mathcal{C}_{seq}\,[\![N \mathbin{..} \mathsf{out}]\!])$$

Run-time system

In the following, we discuss the required procedures in the run-time system. For the sake of presentation and modularity, the procedures are divided into two groups: transformations of the cons-list and administration required for order restoration. Procedures of the former group are discussed in this section, whereas procedures of the latter group are discussed with the compilation schemes discussed in sections 7.3.2 through 7.3.4. The procedure modifyRecord() is not discussed, because at this point, the precise definition is not important and it is replaced in the next compilation scheme.

Procedure `handleInput(`*cont*`)`

```
1 t ← createContainer();
2 markAsFirst(t);
3 WHILE NOT emptyInput()
4 |  t ← insertContainer(cont, t, readInput());
```

The entry-point of the running network is handleInput. It reads records from its input, using the readInput procedure. We do not discuss readInput in further detail, but use it as an abstraction from how records are actually read, buffered and parsed. Every time a record is read from the input, it must be appended to the end of the cons-list. To this end, t always points to the very last (or 'tail') element. The very first container made in handleInput by means of createContainer is the first (or 'head') element of the cons-list. To signify this, the procedure markAsFirst sets the appropriate administrative variables in the container. How this helps to restore the causal order at the output is discussed below.

Procedure `insertContainer`$(cont, c, r)$

```
1 c' ← createContainer();
2 c'_next ↞ c_next;
3 c_next ↞ c';
4 markNextPos(c);
5 c_record ↞ r;
6 IF NOT spawnThread(cont, c) THEN cont(c);
7 ;
8 RETURN c';
```

Until the input is depleted, all following records taken from the input are appended by means of the insertContainer procedure. This procedure takes three arguments: continuation *cont*, container c and record r. Hereby, the container c is the container in which record r must be stored. The return value of insertContainer is a new, empty container, in which a *next* record may be placed. Within the procedure, also administration variables of c are set. A new container c' is created (line 1) and inserted (lines 2–3) between c and its successor c_{next}. Some of the administration variables in c' depend on its predecessor c. This is why the procedure markNextPos (on line 4) is invoked on c (as opposed to c_{next}). It marks the successor of c (c_{next}) as being a successor, i.e. *not* the first container of the cons-list. Like markAsFirst, markNextPos is an order administrating procedure. These procedures are discussed in the conclusions & remarks section below.

Now that c' is set up, r is stored in c and an attempt is made to spawn a thread that will manage the application of continuation *cont* to c. Should there be no resources available for a new thread, before doing anything else, the thread calling insertContainer must perform this task. After either spawning a new thread, or applying the continuation in this thread, insertContainer returns the newly created (empty) container c'.

The last procedure required to obtain a fully functional system is handleOutput. Any thread reaching the output has to wait until isFirst indicates its container is at the head of the cons-list. This only happens when all predecessors have been produced at the output. When it has been established that c is the head of the cons-list, the record in c is produced on the output. Before c is destroyed, its successor c_{next}

Procedure `handleOutput(c)`

```
1 WAIT UNTIL isFirst(c);
2 writeOutput(c_record);
3 propagateFirst(c);
4 freeContainer(c);
```

must be informed that it is now at the head of the cons-list. This is done using the procedure propagateFirst.

Conclusions & remarks

The procedures discussed above handle network in- and output, and define the available transformations on the cons-list. The causal order of records in stateless sequential networks corresponds to the order in which records are read from the input. Because elements are placed, upon input, into the cons-list in-order and no transformations perform any reordering, it follows that the cons-list is at any time a representation of the causal order. Therefore, a record is only produced at the output, if it is in the container at the head of the cons-list (as is the case in handleOutput), it follows that the output order is indeed the causal order.

The procedures that perform position administration (markAsFirst, markNextPos, isFirst, and propagateFirst) are the only occurrences of inter-thread communication. As used above, they can be implemented by having a boolean variable in every container, that is shared between the thread managing that container and the thread managing the container's predecessor. Beyond the initialization of this variable (markAsFirst or markNextPos), the thread managing the container only needs read-access (isFirst), whereas the thread managing the predecessor only needs write access (propagateFirst).

In the following compilation schemes, some complexity is added to these administration procedures. We postpone the precise definition of these procedures until then.

7.3.2 MULTIPLICITOUS BOXES

The first extension to $\mathcal{C}_{seq}$ is to allow multiplicitous boxes. Thus, the result of the application of a box' function to a record can consist of zero-or-more records. When the result consists of a single record, everything works as above, so the two new cases are those where the result is empty, or where it consists of multiple records. When the result is empty, the container should somehow be removed from the cons-list, whereas if there are multiple records in the result, containers should be added. Also, when there are multiple resulting records for which new threads are spawned, all of them must continue through the remainder of the network. In $\mathcal{C}_{seq}$,

this was solved using function composition, which no longer works. Instead, we introduce continuations—as in handleInput—everywhere.

Compilation scheme

We introduce $\mathcal{C}_{mult}$ as a superscheme of $\mathcal{C}_{seq}$, such that it allows for multiplicitous boxes. Because boxes can produce multiple records as a result, the implementation of the box f is now a function of one record to a list of records, i.e. $f : record \to [record]$. Moreover, the compilation of boxes must now take as an argument the continuation to be applied to the results. Thus, for continuation *cont* and container c, we get

$$\mathcal{C}_{mult}\,[\![\mathsf{box}\,(f)]\!] = \lambda cont\,.\,\lambda c\,.\,\mathsf{handleMult}(cont, c, f(c_{record}))$$

where handleMult takes care of any required cons-list transformations.

In the compilation of sequential networks, the compilation of the second network is prefixed to the continuation.

$$\mathcal{C}_{mult}\,[\![N\,..\,M]\!] = \lambda cont\,.\,\lambda c\,.\,\mathcal{C}_{mult}\,[\![N]\!]\ (\lambda c'\,.\,\mathcal{C}_{mult}\,[\![M]\!]\ cont\ c')\ c$$

Because every level of the compilation scheme now expects a continuation, there is no more need for the special out network. The entry-point for $\mathcal{C}_{mult}$ can pass the call to handleOutput as a continuation to the top-level network:

$$\mathcal{C}^{\star}_{mult}\,[\![N]\!] = \mathsf{handleInput}(\lambda c\,.\,\mathcal{C}_{mult}\,[\![N]\!]\ (\lambda c'\,.\,\mathsf{handleOutput}(c'))\ c)$$

Run-time system

The compilation of boxes in $\mathcal{C}_{mult}$ calls the handleMult procedure. The procedure distinguishes three possible cases of the result: zero, one or multiple records.

Procedure `handleMult(`*cont*, *c*, *res*`)`

```
1 SWITCH res DO
2     CASE [ ]
3         markAsDone(c);
4     CASE [r]
5         c_record ↞ r;
6         cont(c);
7     CASE [r : rs]
8         c' ← insertContainer(cont, c, r);
9         handleMult(cont, c', rs);
```

When there are zero results, the container is marked as done and computation on the container ends. Because the container's predecessor and successor may be managed by two different threads, we do not remove the container itself. Instead, it can be removed by the predecessor when it tries to write into variables in this container (i.e. in propagateFirst).

Given a result of precisely one record, that record is stored in the container and the continuation is applied. This behaviour is the same as that of boxes compiled with $\mathcal{C}_{seq}$.

When there are multiple records in the result, new containers have to be inserted into the cons-list. The order in which records are produced in the result is the order in which they must be inserted into the cons-list, which is precisely what insertContainer does.

Conclusions & remarks

Containers may now be added *in between* two containers already in the cons-list. Since every container can be managed by its own thread, we must guarantee not to introduce race-conditions. As observed above, inter-thread communication is limited to threads managing consecutive containers in the cons-list and exclusively in the direction of the cons-list, i.e. a thread managing container c can only send messages to the thread managing $c' = c_{next}$ and the latter receives messages exclusively from the former. A new container $\tilde{c}$ is introduced between two containers, c and $c' = c_{next}$, *exclusively* by the thread managing c. After $\tilde{c}$ is inserted (i.e. $c_{next} = \tilde{c}$ and $\tilde{c}_{next} = c'$), a new thread is spawned to manage c and the thread running handleMult continues with $\tilde{c}$ as its container. Thus, no race-conditions are introduced.

Another advantage of the fact that insertContainer spawns a thread for a container while continuing the calling thread with the successor, is that handleMult can evaluate the list of results lazily. In other words, as soon as the first record of the result is available, a container can be created and a thread spawned for that record before the computation of the remainder of the result continues.

7.3.3 SYNCHROCELLS: LOCAL REORDERING

As described in section 6.2.4, a synchrocell has two or more slots to fill with records. Every slot is filled with the first record (in causal order) to be associated with it. Analogously to how threads arrive out-of-order at the network output, they arrive out-of-order at a synchrocell. Thus, the causal order for records must be (re)established when they arrive at a synchrocell. The following illustrates why the mechanism used for output reordering is too restrictive to use for local reordering at synchrocells.

Consider an empty synchrocell, i.e. no records have arrived yet. We denote the causal order in which records arrive at the synchrocell as a subscript, i.e. the first

record to arrive at the synchrocell is r_1, followed by r_2, etc. Let $\kappa(r_i)$ denote the set of patterns matched by r_i. Furthermore, let S_i denote the set of all patterns matched by records r_j with $j \leq i$, i.e.

$$S_0 = \varnothing$$
$$S_i = \kappa(r_i) \cup S_{i-1}$$

The difference between output reordering and local reordering is that, for local reordering, not *all* predecessors need to have arrived at a synchrocell. If record r_i matches only patterns matched by predecessors, i.e. if $\kappa(r_i) \subseteq S_{i-1}$, then r_i is produced unchanged on the output immediately. In other words, r_i can directly be produced on the output if *some* set of predecessors $P \subseteq \{r_j \mid j < i\}$ exists, which together match all patterns matched by r_i. We call such a set a *κ-satisfying predecessor set*. Formally, a set P is a κ-satisfying predecessor set of record r_i, if and only if

$$\kappa(r_i) \subseteq \bigcup_{p \in P} \kappa(p) \subseteq S_{i-1}$$

The problem is that this causal index i is unknown, because multiplicity can add or remove arbitrarily many records at any point in the stream and multiplicity occurs inside a thread, unbeknownst to other threads. However, the cons-list offers an alternative method of establishing a κ-satisfying predecessor set for record r. Our strategy is to stall r when it arrives at a synchrocell, until either we establish a κ-satisfying predecessor set, or we determine such a set does not exist. In the former case, r continues past the synchrocell, whereas in the latter case, r is stored by the synchrocell.

In order to establish a κ-satisfying predecessor set, threads must be able to communicate to each other the position in the network of the record they manage. We enumerate the (sequential) network, i.e. we give every primitive network an index. Furthermore, we add two new variables to the administration variables of containers: *pos* and *pli*. The variable *pos* indicates the *position* of the container in terms of the index of the last network element that it was output from. The variable *pli* holds the *predecessors' lowest index*, i.e. the minimum *pos* value of all the container's predecessors. Whenever a thread gets output from a network, it increments *pos* in its container. If *pos* was smaller than *pli*, the new minimum of *pos* and *pli* is written in the *pli* variable of the next container in the cons-list.

Before we demonstrate how this helps to relax the task of reordering to that of finding *any* κ-satisfying predecessor set, we illustrate with an example how these extensions allow for local reordering similar to how we performed output reordering for $\mathcal{C}_{seq}$. Consider the network shown in figure 7.1. The network consists of three boxes—with box functions f, g and h, respectively—and one synchrocell—combining a record of type $\{A\}$ with a record of type $\{B\}$. We enumerate this network with the numbers 1 through 4, as shown below the primitive networks in figure 7.1. For the sake of simplicity, all boxes leave the types of records unchanged. Furthermore, all

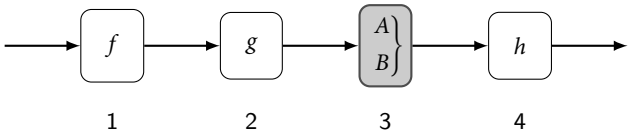


Figure 7.1 – Enumerated sequential network

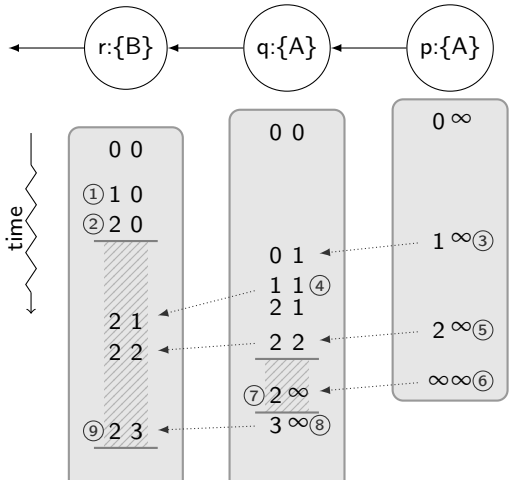


Figure 7.2 – Cons-list with event trace

records flowing into this network have either type $\{A\}$ or type $\{B\}$ and all boxes have both types as input variants. The box at position 4 has type $\{A, B\}$ as an additional input variant.

We input three records—p, q and r—into this network. The first two are of type $\{A\}$, whereas the third is of type $\{B\}$. The output of the record should thus consist of two records: the first $(h \circ g \circ f)(q)$ with type $\{A\}$ and the second $h\left((g \circ f)(p) \cup (g \circ f)(r)\right)$ with type $\{A, B\}$. Figure 7.2 shows, at the top, the initial cons-list. The shaded area below every container is the lifetime of the thread managing that container. We reference these threads by using the name of the record in the container they manage, i.e. thread t_p is the right-most thread in the figure. In every area representing a thread's lifetime, there is a trace (over time) of the values of *pos*, on the left, and *pli*, on the right. The dotted arrows indicate inter-thread communication. All inter-thread communication is asynchronous, so an arrow indicates two separate events: The starting point of the arrow is a send event and the arrow tip is a receive event.

The threads are started in the order of arrival of their corresponding records. The

first container was given a *pli* value of ∞ by markAsFirst. All other containers start with both *pos* and *pli* set to 0. Events labelled with an encircled number are as follows:

1. t_r finishes applying f to its record and increments the *pos* field of its container.
2. t_r finishes applying g to its record, increments its container's *pos* and arrives at the synchrocell. At this point, t_r can not decide whether its record will be consumed by the synchrocell. Thus, t_r waits. The waiting period is indicated with the area shaded with diagonal lines.
3. t_p finishes applying f to its record, increments it container's *pos* and sends a new *pli* value to its successor, t_q.
4. t_q finishes applying f to its record, increments the *pos* field of its container and sends a new *pli* to t_r.
5. t_p finishes applying g to its record, increments its container's *pos*, sends a new *pli* to t_q and arrives at the synchrocell. Since the synchrocell has index 3 and $pli = \infty \geq 3$, t_p can decide that all predecessors have already passed this synchrocell.
6. The record in p is stored in the synchrocell. Having no more record, t_p is finished, making its $pos = \infty$. It sends a new *pli* to t_q and terminates.
7. t_q receives the value ∞ for its *pli*. Thus, all predecessors have passed this synchrocell, because $pli = \infty \geq 3$. t_q tests to see whether unmatched patterns remain in the synchrocell.
8. No patterns matched t_q's corresponding record, so it is output by the synchrocell and continues to network 4, the box with function h.
9. t_r receives the value 3 for its *pli*. Thus, all predecessors have passed this synchrocell, because $pli = 3 \geq 3$, so t_r can test for unmatched patterns.

The above example shows that the use of the *pos* and *pli* variables suffices to locally restore causal order. As discussed above, the problem can be further relaxed to finding a κ-satisfying predecessor set. To do this, we introduce shared variables to represent the synchrocell. Simultaneous access to these variables is prevented by a mutually exclusive locking mechanism. When a thread wants access to the variable, it requests a lock, waits until it is granted the lock (at which point the thread is guaranteed exclusive access), performs its critical section and releases the lock. The model we use for the implementation of these locks is a read-on-lock, release-on-write model, i.e. reading a shared variable automatically requests the lock and writing to the variable automatically releases the lock. For every pattern p, we require one variable, $plimax_p$, that holds the highest *pli* value of all containers containing records that match p that have arrived at the synchrocell. When a thread at a synchrocell finds that the *plimax*-value for every pattern matched by its record is larger than its own *pli*-value, it has established the existence of a κ-satisfying predecessor set.

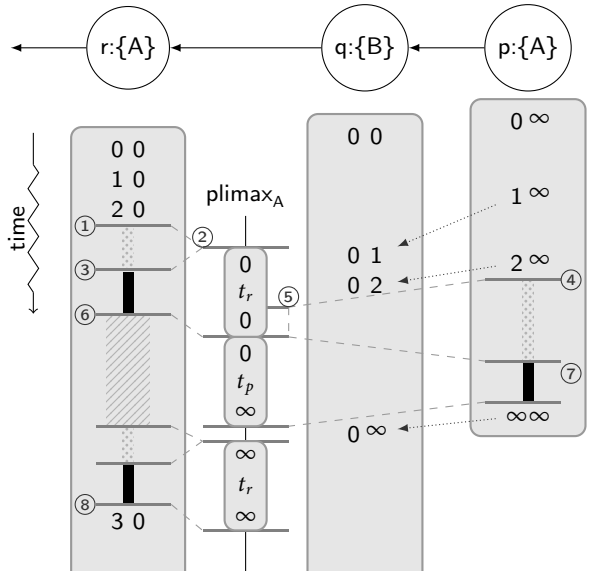


Figure 7.3 – Cons-list and synchrocell variable with event trace

To be able to properly demonstrate how this relaxes the reordering, we adapt the above example. For the same network, we now use the input shown in figure 7.3, i.e. the types of q and r have been swapped. Furthermore, the variable $plimax_A$ has been added to the figure. This is the variable corresponding to the $\{A\}$ pattern in the synchrocell. The shaded areas between two horizontal bars indicates the variable is locked. For every locked period, from top to bottom, the initial value, the owner of the lock and the final value are shown. The dashed lines to and from the horizontal bars of the variable represent the lock negotiations (request, acquire and release). Again, we discuss a few events, as numbered in the figure.

1. t_r is the first to reach the synchrocell and requests a lock for the variable corresponding to the matching pattern $\{A\}$: $plimax_A$. Until the lock is acquired, t_r stalls (the dotted bar).
2. The variable is locked for t_r and the value contained in $plimax_A$ (0) is sent to t_r.
3. t_r receives the lock and the value from $plimax_A$, so it enters its critical section (the black bar).
4. t_p reaches the synchrocell, matches pattern $\{A\}$ and requests a lock for the corresponding variable.
5. The variable is locked by t_r, so the lock request from t_p is stalled.
6. After acquiring the lock, t_r calculated the maximum of its own *pli* and the value received from $plimax_A$. The result is 0, which is sent back to $plimax_A$.

This releases the lock on $plimax_A$. Because the result is smaller than the synchrocell's index, no predecessors with a record with type $\{A\}$ have yet been found. Thus, t_r sleeps (the striped area) before trying again.

7. t_p has received the lock on and value of $plimax_A$. It computes the maximum of $plimax_A$'s value and that of its own *pli* and returns the result (∞). Since the result is larger than the synchrocell's index, the thread decides record p is consumed by the synchrocell. Thus, after releasing the lock, it terminates as in the previous example.
8. When trying $plimax_A$ again, t_r has found a value in $plimax_A$ that is higher than t_r's own *pli*. This means that there exists a predecessor that matches this pattern. Because this is the only pattern matched by r, t_r has found a κ-satisfying predecessor set. Therefore, it may continue beyond the synchrocell.

Compilation scheme

The compilation scheme $\mathcal{C}_{sync}$ is a superscheme of $\mathcal{C}_{mult}$. It only adds a definition for synchrocells. Let $\mathsf{sync}\,(\kappa)$ denote a synchrocell, where κ is a function from records to the set patterns matched by the record, i.e. $\kappa : record \to \wp pattern$. Incrementing the *pos* variable after every box is dealt with by passing to handleMult, which is used to process a box' results, a container with an incremented *pos*. The entire compilation scheme $\mathcal{C}_{sync}$ is as follows:

$$\mathcal{C}_{sync}\,[\![\mathsf{box}\,(f)]\!] = \lambda cont\,.\,\lambda c\,.\,\mathsf{handleMult}(cont, \mathsf{posInc}(c), f(c_{record}))$$
$$\mathcal{C}_{sync}\,[\![\mathsf{sync}\,(\kappa)]\!] = \lambda cont\,.\,\lambda c\,.\,\mathsf{handleSync}(cont, c, \kappa(c_{record}))$$
$$\mathcal{C}_{sync}\,[\![N\,..\,M]\!] = \lambda cont\,.\,\lambda c\,.\,\mathcal{C}_{sync}\,[\![N]\!]\ \left(\lambda c'\,.\,\mathcal{C}_{sync}\,[\![M]\!]\ cont\ c'\right)\ c$$
$$\mathcal{C}^{\star}_{sync}\,[\![N]\!] = \mathsf{handleInput}(\lambda c\,.\,\mathcal{C}_{sync}\,[\![N]\!]\ (\lambda c'\,.\,\mathsf{handleOutput}(c'))\ c)$$

Run-time system

There are two new procedures required of the run-time system: handleSync and posInc. The latter is an administrative procedure, discussed in the next section. handleSync implements the behaviour explained above and is discussed next.

The procedure handleSync consists of two parts. The first part determines whether or not a κ-satisfying predecessor set exists (lines 4–15). The second part determines whether the result of the synchrocell is stored in this thread's container (c), or in one of its successors (lines 16–31). In order to prevent race-conditions in both parts, there is never more than one lock on a shared variable required to proceed. Furthermore, because c_{pli} is asynchronously updated by the calling thread's predecessor, a local copy is kept (line 3) that is only updated at the end of each loop body (lines 13–15 and lines 28–30). Before the local *pli* is updated, the successor's *pli*-value is updated, using propagatePli. This way, if c_{pli} gets updated right after propagatePli, the local variable has a newer (and thus higher) value than the successor.

Procedure `handleSync(`*cont*, *c*, *M*`)`

```
1  s ← getSyncState(c_pos);
2  H ← ∅;
3  pli ← c_pli;
4  WHILE M ≠ ∅  FOREACH p ∈ M DO
5  |  plimax ← getVariable(s(p))                    ▷ 1, 2 & 3 in figure 7.3;
6  |  IF plimax > pli THEN                           ▷ 8 in figure 7.3
7  |  |  M ← M ∖ {p};
8  |  ELSE IF pli > c_pos THEN                       ▷ 7 in figure 7.3
9  |  |  M ← M ∖ {p};
10 |  |  H ← H ∪ {p};
11 |  |  plimax ← ∞;
12 |  setVariable(s(p), max(pli, plimax))           ▷ 6 in figure 7.3;
13 |  IF c_pli > pli THEN
14 |  |  propagatePli(c);
15 |  |  pli ← c_pli;
16 IF H ≠ ∅ ∧ ¬matchesAll(s, H) THEN
17 |  REPEAT
18 |  |  done ← false;
19 |  |  plimin ← getVariable(s_output);
20 |  |  IF isComplete(s, H) THEN
21 |  |  |  c_record ↞ combineSyncMatches(s, H, C_record);
22 |  |  |  done ← true;
23 |  |  ELSE IF pli > plimin THEN
24 |  |  |  storeRecord(s, H, c_record);
25 |  |  |  markAsDone(c);
26 |  |  |  done ← true;
27 |  |  setVariable(s_output, min(pli, plimin));
28 |  |  IF c_pli > pli THEN
29 |  |  |  propagatePli(c);
30 |  |  |  pli ← c_pli;
31 |  UNTIL done ;
32 IF ¬isDone(c) THEN
33 |  posInc(c);
34 |  cont(c);
```

The set M contains all patterns matched by the record in c. For every pattern $p \in M$, we try to determine whether a predecessor matches it or not. The critical section (lines 5–12) works as described in the example above; If a higher *pli*-value than the local *pli* has been stored in the variable (line 6), a predecessor has been found that matches pattern p. If this is not the case and all predecessors have passed the synchrocell (line 8), this thread's record is the first (in causal order) to match the pattern. In either case, pattern p no longer needs to be tested, so it can be removed from M. In the latter case, p is stored in the set of 'hits' H and, by setting *result* to ∞, it is guaranteed that no successor can conclude it has found a hit for p.

A synchrocell is defined to produce a result for the record that matches all remaining (in causal order) patterns. Thus, if a hit was found (line 16), the thread calling handleSync must decide whether it is the one that carries on with the result. The procedure matchesAll catches the corner case where one record matches all patterns, in which case the calling thread simply continues past the synchrocell. The last arriving container has the lowest *pli*-value, so now, opposite to the above, the *minimum pli* should be found. This is done in lines 17–31. The variable s_{output} is initialized to the synchrocell's position. The procedure isComplete is true if all slots, except those indicated in its second argument, have been filled by calls to storeRecord. If no successor was found and all slots have been filled, the calling thread must continue with the combined result (lines 20–22). If a successor is found (i.e. a lower *pli*-value is stored in s_{output}), the thread stores its record in the synchrocell, clears up the container and finishes execution by *not* applying the continuation to the container (lines 23–26).

Conclusions & remarks

Besides being used for local reordering at synchrocells, the *pli* variable is also used to implement output reordering: markAsFirst(c) sets c_{pli} to ∞ (or the highest value of the type representing positions), markNextPos(c) sets the *pli* of the successor to $\min(c_{pli}, c_{pos})$, isFirst(c) tests whether $c_{pli} = \infty$ and propagateFirst is reduced to a specific case of propagatePli.

At any point in time, the *pli*-values are weakly decreasing from the head of the cons-list to the tail, because every thread passes on to its successor the minimum of its own *pli* and its position. Since records can only move forward through the network, their own position is weakly increasing over time. This means that the *pli*-value of a single container is also weakly increasing over time.

$\mathcal{C}_{sync}$ and the accompanying run-time system do not introduce new transformations of the cons-list. As before, insertContainer is the only procedure to introduce new containers (and spawn threads). It is still only called by handleInput and handleMult. When threads die, they still markAsDone their container and let their predecessor clean it up (now in propagatePli).

$$
\begin{aligned}
\mathcal{C}_{hydra}\,[\![\mathsf{box}\,(f)]\!] &= \lambda cont\,.\,\lambda c\,.\,\mathsf{handleMult}(cont, c, f(c_{record})) \\
\mathcal{C}_{hydra}\,[\![\mathsf{sync}\,(\kappa)]\!] &= \lambda cont\,.\,\lambda c\,.\,\mathsf{handleSync}(cont, c, \kappa(c_{record})) \\
\mathcal{C}_{hydra}\,[\![N\,..\,M]\!] &= \lambda cont\,.\,\lambda c\,.\,\textsc{let} \\
&\qquad cont_M = cont \circ \lambda c'\,.\,\mathsf{posQes}(c') \\
&\qquad cont_N = cont_M \circ \mathcal{C}_{hydra}\,[\![M]\!] \circ \lambda c'\,.\,\mathsf{posInc}(c') \\
&\quad \textsc{in}\ \mathcal{C}_{hydra}\,[\![N]\!]\ cont_N\ (\mathsf{posSeq}(c)) \\
\mathcal{C}_{hydra}\,[\![N \parallel^{\sigma}_{\delta} M]\!] &= \lambda cont\,.\,\lambda c\,.\,\textsc{let} \\
&\qquad cont' = cont \circ \lambda c'\,.\,\mathsf{posTla}(c') \\
&\qquad left = \lambda c'\,.\,\mathcal{C}_{hydra}\,[\![N]\!]\ cont'\ \mathsf{posAlt}(c', 0) \\
&\qquad right = \lambda c'\,.\,\mathcal{C}_{hydra}\,[\![M]\!]\ cont'\ \mathsf{posAlt}(c', 1) \\
&\quad \textsc{in if}\ \sigma(c_{record}) = 0\ \textsc{then}\ left\ \textsc{else}\ right \\
\mathcal{C}_{hydra}\,[\![N^{\star}_{\delta,\gamma}]\!] &= \lambda cont\,.\,\lambda c\,.\,\textsc{let}\ cont' = \lambda rec\,.\,\lambda c'\,.\,\textsc{if}\ \gamma(c') = recurse \\
&\qquad \textsc{then}\ rec\ (\mathsf{posRepl}(c')) \\
&\qquad \textsc{else}\ cont\ (\mathsf{posReti}(c')) \\
&\quad \textsc{in}\ \mathbf{Y}\ cont'\ (\mathsf{posIter}(c)) \\
\mathcal{C}_{hydra}\,[\![N\,!_{\delta}\,t]\!] &= \lambda cont\,.\,\lambda c\,.\,\textsc{let} \\
&\qquad cont' = cont \circ \lambda c'\,.\,\mathsf{posTla}(c') \\
&\qquad c' = \mathsf{posAlt}(c, c_{record}[t]) \\
&\quad \textsc{in}\ \mathcal{C}_{hydra}\,[\![N]\!]\ cont'\ c' \\
\mathcal{C}^{\star}_{hydra}\,[\![N]\!] &= \mathsf{handleInput}(\lambda c\,.\,\mathcal{C}_{hydra}\,[\![N]\!]\ \left(\lambda c'\,.\,\mathsf{handleOutput}(c')\right)\ c)
\end{aligned}
$$

FIGURE 7.4 – The final compilation scheme

7.3.4 THE FINAL SCHEME

The final scheme is called $\mathcal{C}_{hydra}$. It is a superscheme of $\mathcal{C}_{sync}$, adding parallel composition, serial replication and inspection combination. These three combinators share the property that networks inside them can not be enumerated as networks compiled with $\mathcal{C}_{sync}$. For example, whereas in $N\,..\,M$ it was clear that network N has index 1 and network M has index 2, networks in $N \parallel_{\delta} M$ can not be enumerated, because N does not appear *before* M, nor vice versa. Consequently, when there is a synchrocell inside a network in a parallel composition, we need an alternative way to establish κ-satisfying predecessor sets. We do this by changing the indexing of networks. For $\mathcal{C}_{sync}$, we enumerated networks with fully ordered indices. Instead, we use the network indices discussed in section 6.4.3 and section B.3. These network indices are partially ordered. It follows that the *pli*-value that is propagated to a successor is no longer the *minimum* of a container's position and its own *pli*, but rather the *infimum* of the two.

The final scheme is shown in figure 7.4. All procedures with a 'pos' prefix are procedures that change the position index. For every network index element (section B.3), there are two corresponding procedures: one to add the network index element to the index stored in c_{pos} and one to take the network index element away again. The former has the network index element name as a suffix, whereas the latter has the reversed network index element name as a suffix. This leaves posInc and posRepl. posInc was used before, but now it is specialized to increment the least-significant network index element, which must be a `Seq`. posRepl does the same for `Iter` network index elements (used for the serial replication). All procedures adding or incrementing network index elements automatically call propagatePli.

The $\mathcal{C}_{hydra}$ compilation scheme ignores the δ annotations of all operators. For $\mathcal{C}_{hydra}$ all mergers are deterministic. Non-determinism exists in SNet to relax the order constraint on records in a stream. The execution model in `snetc` uses this explicitly to perform first-come first-serve mergers, i.e. the non-deterministic choice of which record from the input streams to place in the output stream is based on earliest availability in `snetc`. Hydra's execution model does not benefit from this relaxation, because *all* processing already occurs out-of-order.

7.4 No introduction of non-termination

In this section, we analyse Hydra's execution model with regards to termination behaviour. It is possible to write non-terminating networks in SNet. Therefore, non-termination for every SNet network can not be decided. This problem is known as the halting problem. For networks that *can* be shown to terminate for every input, however, we give a strong indication that Hydra does not introduce non-termination due to deadlock or starvation.

Network termination means, that for a finite input, with a finite number of computations, a finite output is produced and all records that remain in the network are stored in synchrocells. For every network that is guaranteed to reach this state, executing it with Hydra will reach this state. In the following, we can demonstrate that the container at the head of the cons-list always reaches its end (either at the network output, being consumed by a multiplicity of zero or by being stored in a synchrocell). Furthermore, we demonstrate that its successor becomes the new head. Since network termination implies a finite output result, by induction these results imply that the complete output is produced. Thus, Hydra introduces no starvation. Lastly, we need to show that no threads can deadlock.

7.4.1 starvation

After the container at the head of the cons-list reaches its end, markAsDone propagates its *pli*-value to its successor. The container at the head of the cons-list is the only container with a *pli* value of ∞. Therefore, when a container c with $c_{pli} = \infty$ is

passed to markAsDone, it can be removed and freed, indeed making the successor the head of the cons-list. Thus, it only remains to show that a container with a *pli*-value of ∞ always reaches its end. To show this, it suffices to show two properties: Firstly, that if there is a thread processing the container at the head of the cons-list, it reaches the container's end. Secondly, that there is always a thread processing the container at the head of the cons-list.

For the first property, observe that containers flow unobstructed through every type of network, except the synchrocell. When the container at the head of the cons-list reaches a synchrocell, it either matches at least one pattern[2], or bypasses the synchrocell (e.g. if it has already synced). Because only the container at the head of the cons-list has a *pli*-value of ∞, the test for predecessors (line 6 of handleSync) always fails. Because all network indices are finite, the test to decide non-existence of predecessors (line 8) always succeeds. Similarly, the test for the existence of successors (line 23) always succeeds. This demonstrates that the container at the head of the cons-list has an unobstructed flow through the network.

For the second property, we examine the creation of threads. Initially, there is only one thread that processes handleInput. Threads are created by spawnThread. This procedure is only called from insertContainer. insertContainer inserts a new container *after* the container given as an argument. It tries to spawn a thread for the container of its argument and return the newly inserted container. If a thread is successfully spawned, this new thread continues with the container from the argument. When spawning a thread fails, the continuation is applied to the container from the argument, *before* returning the newly created container. Thus, when insertContainer is passed the container at the head of the cons-list, there is always a thread to continue processing that container before processing any other. insertContainer is called from only two procedures: handleInput and handleMult. Neither of these apply a continuation to a container before calling insertContainer. This demonstrates that there is always a thread processing the container at the head of the cons-list.

7.4.2 DEADLOCK

Deadlock occurs when two or more threads are waiting for each other to release a lock. In Hydra, threads share only two types of variables: *pli*-variables and state variables of synchrocells.

A container c managed by thread t_c has a successor c', pointed to by c_{next}, i.e. $c_{next} = c'$. Container c' is either managed by the same thread (if spawning a thread for c failed when c' was inserted), or it is managed by another thread $t_{c'}$. In the former case, there is no sharing. In the latter case, t_c and $t_{c'}$ share the variable c'_{pli}. However, t_c only has write-access and $t_{c'}$ only has read-access. Because the

[2] The type-based routing of records enforces that every record that reaches a network matches at least one of the network's input variants (see [37]).

communication through c'_{pli} is asynchronous, it is implemented as a lock-less variable. Therefore, it can not be (part of) the cause of deadlock.

The synchrocell state variables *do* require locking. There are two critical sections in handleSync: lines 5–12 and lines 19–27. Both critical sections require only *one* lock. Therefore, no circular dependencies between threads requesting locks can arise. Furthermore, the computations inside the critical sections are strictly terminating. This demonstrates no deadlocks are introduced.

7.5 Conclusion

In this chapter, two problems were identified with the execution model implemented in the current SNet-compiler, `snetc`. A new implementation, Hydra, with a new execution model was discussed. The required compilation scheme and run-time system were developed. The problems observed in the execution model of `snetc` are solved by Hydra: records are processed out-of-order, unrolling of serial replication no longer poses a resource management problem and the communication between distributed memory domains has been reduced to synchronisation messages. Also, a strong indication is given that Hydra does not introduce non-termination for networks and input that terminate according to the semantics of SNet.

Conclusion

Conclusions & recommendations

8.1 On-line spatial resource management

Applications for embedded systems and the environments in which these embedded systems have to operate are both continuously widening. This increases the complexity of such systems, leading to higher development costs. Higher costs and an ever more critical time-to-market pushes system design towards modularization of both hard- and software. This modularization often leads to what is referred to in this thesis as tiled systems. Added benefits of tiled systems include easy scalability and yield increase. Both are achieved by adding more of the same 'tiles' to a design. This flexibility of the design does come at the cost of making it harder to predict resource management at design-time. Opening up such systems to applications introduced by the user makes the application set highly unpredictable at design-time as well. The unpredictability has become prohibitive and demands heuristics for resource management. In our opinion, it even forces the move of (at least part of) resource management from design-time to run-time.

the work presented in this thesis

Solutions found in the literature and discussed in chapter 2 all solve (slightly) different problems. We have given a formalization (in chapter 3) of the larger problem that encompasses all problems identified by the related work. For this formalization, we have given formal definitions of the hardware components (HwEs, routers, links and interfaces) that together form a platform and of the software components (tasks,

implementations and channels) that together form an application. We have given definitions for resource capacities and resource requirements, that are each other's dual, such that they can be compared to see whether the resource requirements of a software component are met by the resource capacities of a hardware component. Furthermore, we have provided ways to composite, accumulate and threshold both capacities and requirements. Finally, we have defined QoS constraints on applications.

With these definitions, we have given definitions for what constitutes on-line resource management, i.e. finding an assignment of applications to platforms that meet the application's QoS constraints. Such assignments, called execution layouts, have been given qualitative metrics (adherence, adequacy and feasibility) and quantitative metrics (cost), so that they can be compared.

We have introduced (in section 3.4) and implemented (chapter 4) a heuristic, that may be used to perform on-line resource management and shown how it relates to the qualitative metrics. The implementation has been described on a conceptual level by algorithms and their analysis, and on a concrete level by means of an implementation in a Linux kernel, named Kairos, providing user interfaces that adhere mostly to common practices. Furthermore, the input required by the on-line spatial resource manager has been specified and intuitions have been offered to designers of applications on how to interpret abstract concepts in this input.

By means of a large case study and a synthetic benchmark, we have demonstrated that Kairos is applicable in realistic scenarios in chapter 5. The time it requires to find execution layouts is well within the acceptable limits for the intended platforms and applications. The solutions found compare relatively well to exhaustively optimized solutions.

Kairos' performance (both in computational terms, as well as in terms of the result) improves significantly when applications are (at least) an order of magnitude smaller than the platform on which they have to run. Heterogeneity helps to reduce the execution cost of applications, both in low and high utilization cases. On-line spatial resource management can deal with heterogeneity, but if such flexibility is desired, the design of platforms should take into consideration some additional constraints, i.e. heterogeneity is best distributed evenly over the platform and the interconnect should offer sufficient resources.

Recommendations for future work

On-line spatial resource management vastly increases a system's flexibility and, therewith, its applicability. If standards for platform design can be developed (either by formal standardization organizations or by specialized industrial consortia), this new flexibility may lead to a considerable life-time extension of embedded systems as application platforms. In other words, it may do for the mobile phone what x86 backwards-compatibility did for the desktop computer: Make software for

the current generation of hardware immediately accessible for the next generation. This is why standardization and a further exploration of the greatest common denominator of such systems is worth pursuing.

In that same light, hardware design can be improved for this new perspective on embedded systems. HwEs are still commonly developed either for the general purpose or for specific scenarios, e.g. assuming very specific integration in a hardware platform. Design methods should be developed that make HwEs even more off-the-shelf components for system design. For example, processors should have local instruction memories large enough to store typical software kernels, to lower latency, communication volume and dependency on interconnect architecture.

More specifically for on-line resource management, better benchmarks are required. Both synthetic benchmarks and databases of case studies can serve as a driver for research and the exploration of new programming paradigms that take on-line resource management into account. Such benchmarks lead not only to better comparisons of resource managers, but also expose strengths and weaknesses in the design of applications and platforms.

The added flexibility offered by on-line resource management also increases the desirability of having systems capable of task migration. This is a challenge for all disciplines involved in embedded system design: from hardware designer to real-time analyst. Allowing task migration will open doors to ad-hoc adding and removing of hardware, but also to switching power modes and QoS levels.

8.2 SNet

We have given a denotational semantics of the language SNet (in chapter 6) by giving a translation from SNet to the pure, lazy, functional programming language Haskell. In this translation, non-determinism in SNet has been translated to choices represented by an oracle. Given an oracle, the behaviour of a translated SNet network is purely functional, i.e. fully deterministic. Using the presented denotational semantics, a proof has been provided that, for every oracle, every SNet network is prefix monotonic.

Furthermore, two problems were identified with the execution model implemented in the current SNet-compiler, `snetc`. A new implementation, Hydra, with a new execution model was discussed in chapter 7. The required compilation scheme and run-time system were developed. The problems observed in the execution model of `snetc` are solved by Hydra: records are processed out-of-order, unrolling of serial replication no longer poses a resource management problem and the communication between distributed memory domains has been reduced to synchronisation messages. Also, Hydra has been shown to not introduce non-termination for networks and input that terminate according to the semantics of SNet.

SNET separates the concerns of application engineering and concurrency engineering. This was motivated by a desire to relieve the application engineer from the responsibility of concurrency engineering. The downside of this approach is that it makes it harder for the application engineer to convey the knowledge he or she *does* have about concurrency opportunities in the application. For example, because SNET assumes possible data dependencies everywhere, the characterization of the input data can have a large impact on the performance of a running SNET network. The possibilities for optional input from the application engineer must be further explored.

The characterization of (input) data and workload can be further exploited by adding to a run-time system a means of profiling and (possibly) just-in-time compilation. Resource management may also benefit from such run-time profiling. We identify this (run-time profiling based resource management) as the most promising direction of future research.

Benchmark results

This appendix lists results for the benchmarks discussed in chapter 5. First, we present the Kairos configurations used. Next, we report on run-times of the experiments.

A.1 Kairos configurations

Table A.1 shows the four configurations of Kairos used. The parameters refer to those used in chapter 4. The k parameter represents the added search depth in the search for a covering HwE set, as used in algorithm 2. The parameters c and f indicate the weights used in the cost function for communication and fragmentation reduction, respectively (see algorithm 4). The same cost function uses a large constant to indicate the cost of traversing an unknown distance. This is parameter U.

Optimization target	Name	k	c	f	U
None (First-Fit)	FF	2	0	0	1000
Communication Cost	CC	2	1	0	1000
Reduce fragmentation	RF	2	0	1	1000
Comm. cost / red. frag.	CF	2	2	1	1000

Table A.1 – Kairos configurations

Request	Platform			
	MeshHo	MeshHe9	MeshHe94	MeshHeClust
A1	3180	5120	2290	1780
A2	25850	6200	1980	4540
A3	8250	4350	3960	3630
A4	1410	3370	1610	920
rst				
A4		2400	1050	1430
A3		4130	2990	5050
A2		1250	1550	1990
A1		1830	1250	1120
Clairvoyant	202935360	3953550	87080	153470

Table A.2 – Run-times [ms] for the ILP solutions

A.2 Run-times

The run-times of the ILP solution are shown in table A.2. At the bottom are run-times for the clairvoyant solution. As discussed in chapter 5, the ILP solution does not take into account validation.

For space reasons, we only list the run-times of two Kairos configurations (in table A.3). The variation of these results for the other two configurations was minor. Because of the results are in the microsecond scale (whereas the ILP results are expressed in milliseconds), they are more sensitive to small variations in the system. In our experimental set-up, it was not possible to measure accurately the effect of interrupt handling by the Linux kernel.

The ILP solution was calculated using CPLEX version 12.0.1 on a server with two Intel Xeon E5430 (quad-core) processors at 2.66 GHz. Assigned resources were limited at four cores and two gigabytes of memory. Utilization of the four assigned cores was continuously over 95% for the duration of the tests. The Linux kernel with the Kairos implementation was run on an ARM926 processor at 200 MHz.

		MeshHo				MeshHe9				MeshHe94				MeshHeClust			
	Req.	B	M	R	V	B	M	R	V	B	M	R	V	B	M	R	V
First-Fit	A1	320	11941	1935	4375	235	10451	1955	29012	414	15895	1985	4945	204	13357	2177	7575
	A2	413	10198	2316	6159	334	11560	2317	5762	289	12350	2302	5677	277	14819	2539	5628
	A3	714	16579	3149	11854	515	15479	3385	9650	427	17238	3182	11432	400	19160	3099	11158
	A4	349	15599	1461	5230	254	18695	1704	4602	228	9476	1522	7411	220	11847	1752	4266
	rst																
	A4	295	10711	1585	4318	261	46761	1526	5400	225	16862	1769	4598	258	16852	1913	5398
	A3	669	16781	3142	11531	499	15193	3302	10298	414	15709	2932	12681	384	18742	3080	14590
	A2	929	11880	3392	5733	346	12739	2253	5623	302	12970	2579	5636	283	13866	2957	5760
	A1	317	11367	1871	3824	261	11201	2180	5702	257	5822	3227	5144	220	10010	2042	7352
Comm./Frag.	A1	298	10488	2102	7225	269	11211	2516	13731	331	11811	2405	5340	198	14907	2287	5231
	A2	407	12437	2399	7192	378	12828	2556	8787	305	11409	2322	7813	293	12548	2919	9075
	A3	712	21310	3293	8628	627	18257	3170	9135	446	16461	3452	9846	400	18330	3058	13854
	A4	321	15697	1625	4541	283	15866	1922	8459	250	10414	1586	4720	214	11004	1828	4310
	rst																
	A4	291	11246	1632	4514	256	15485	1508	4750	222	17838	2058	22334	233	17311	1906	5417
	A3	672	16415	3511	15495	552	15630	3410	18487	469	17408	3124	9101	387	19164	3177	7525
	A2	427	13880	2419	7211	578	13593	2260	5929	382	15305	2807	5925	289	13199	2510	5625
	A1	331	13480	2088	5116	283	11458	2001	5092	289	5401	3195	5234	220	10312	2042	7710

Table A.3 – Run-times [µs] of two representative Kairos configurations (fragmentation and mixed communication/fragmentation) per platform, per phase: (B)inding, (M)apping, (R)outing, (V)erification

STRUCTURE DEFINITIONS FOR SNET

B.1 CORE REPRESENTATION OF SNET

```
{-# LANGUAGE RecordWildCards, TypeSynonymInstances #-}
module Language.SNet.Syntax where

import qualified Data.IntMap as IM
import qualified Data.IntSet as IS
import           Data.Array.IArray hiding ((!))
import qualified Data.Array.IArray as    A((!))

type ℕ   ≙ Int
type ℤ   ≙ Int
type ℘ℒ ≙ IS.IntSet
```

Labels in records are encoded as integers for fast access. Possibly, the original names can be stored in lookup-tables in the context, but in general, we do not want strings in the run-time environment. For fast resolution of fields, tags and binding tags, they are encoded as follows:

```
type ℒ ≙ ℤ

isField, isTag, isBindingTag :: ℒ → Bool
isField      n ≙ n > 0
isTag        n ≙ n < 0 ∧ even n
isBindingTag n ≙ n < 0 ∧ odd  n
```

The types for record content are either tag values ($\mathcal{T}$) or box language values (`blv`). Because the difference between fields and tags is abstracted away from Haskell's type system by encoding them all as ints, the content of a member of a record is either an instance of $\mathcal{T}$ or of `blv`. This gives the general 'content type' for any record member: $\mathcal{C}$.

```
type 𝒯 ≙ ℤ
type 𝒞 ≙ Either blv 𝒯
```

Now the representation of the box type ($\mathfrak{B}$) and box function ($\mathcal{F}$) can be given. Note that a box can produce records of different *variants* of record types. In this implementation, every response form a box is the content of the response, annotated with the variant number. This variant number is the index in the range of the box type, viz.

```
data 𝔅 ≙ [ℒ] ↦ Array ℕ [ℒ]
type BoxFunc blv ≙ [𝒞] → [(ℕ, [𝒞])]
```

Record types ($\mathfrak{R}$) are described by the total set of labels contained in a record of that type. In other words, the Haskell type for record types is a set of labels. Given a box language value type `blv`, a record is a finite map from labels to their corresponding content. Thus, the empty record is an empty map and the type of a record is the domain of the finite map. Extracting a member from the record is a projection from the label to the content.

```
type ℜ ≙ ℘ℒ
type ℛ ≙ IM.IntMap 𝒞

emptyRec ≙ IM.empty    :: ℛ
typeOf   ≙ IM.keysSet :: ℛ → ℜ

(!) :: ℛ → ℒ → 𝒞
(!) ≙ (IM.!)

combine :: ℛ → ℛ → ℛ
combine ≙ IM.union
```

A record must be converted to a list of relevant content, before a box function can be applied to it. Likewise, the output of a box must be translated back into a record. The former is simply the sequential selection of all members from the record in the order in which their labels occur in the domain of the box type. The latter consists of first selecting the variant indicated in the result and then using that variant to convert the contents of the result into a record.

```
toBox :: 𝔅 → ℛ → [𝒞]
toBox (dom ↦ _) r ≙ map (r !) dom

fromBox :: 𝔅 → (ℕ, [𝒞]) → ℛ
fromBox (_ ↦ ran) (i,es) ≙ IM.fromList $ zip (ran A.! i) es
```

SNet defines a sub-/supertype relation on record types. The relation is defined such that both of the following rules must hold:

1. All binding tags in the supertype must also be in the subtype and vice versa.
2. The subtype is a superset of the supertype.

```
(⩽) :: 𝔑 → 𝔑 → Bool
sub ⩽ super ≙ subbinds = superbinds ∧ supermisc ⊆ submisc
  where
  (subbinds,   submisc  ) ≙ IS.partition isBindingTag sub
  (superbinds, supermisc) ≙ IS.partition isBindingTag super
```

Network types are formulated on records. The simple network type ($\mathfrak{F}$) has a single record type as a domain and all possible response record types as a range. A full network type ($\mathfrak{N}$) is a collection of all variants of simple network types that together describe the full variance of the network. Box types can be generalized to simple network types by translating the lists of types of content to record types.

```
data 𝔉 ≙ 𝔑 ↣ [𝔑]
type 𝔑 ≙ [𝔉]

generalize :: 𝔅 → 𝔉
generalize (dom ↦ ran) ≙ IS.fromList dom ↣ map IS.fromList (elems ran)
```

Routing through a network is based on type matches. Because types can overlap (sub-/supertypes), preference for one route over another is given by means of the 'strenght' of a match. The strength of the match between network type t and record r is -1 if the type of r is not a subtype of t. Otherwise, it is the size of the largest matching variant of t.

```
match :: ℛ → 𝔑 → ℤ
match rec net ≙ maximum
    (-1 : [ IS.size dom | (dom ↣ _) ← net, typeOf rec ⩽ dom ])
```

Another important concept from SNet that involves both routing and type is flow inheritance. When a record is fed into a box, the members of the record that the box does not take as input are flow inherited. This is done in separate steps here: separation ($\prec$) and fusion ($\succ$). Based on the expected input type of a network, a record can be separated into a 'through part' and an 'around part.' These are non-overlapping records that, together, make up the original record. To not have to a lot of type plumbing in any implementation using this module, the separation is a type class, the members of which are the different types. Separation based on simple network types is the same as separation on the domain of the simple network type. Separation on box types is the same as separation on the generalized box type. The important definition is that of separation based on record types. For fusion, it is important to note that for any members that occur both in the flow inherited set *and* in the result, the members in the result are preferred. Haskell map unions prefer elements in the left operand.

```
data 𝔍 ≙ 𝓘 { through, around :: ℛ }

class    Inheritable t where (≺) :: t → ℛ → 𝔍
instance Inheritable 𝔉 where (≺) (t ↣ _) ≙ (≺) t
instance Inheritable 𝔅 where (≺)         ≙ (≺) ∘ generalize
instance Inheritable 𝔑 where
  (≺) t ≙ uncurry 𝓘 ∘ IM.partitionWithKey (λk _ → k ∈ t)

(≻) :: 𝔍 → ℛ → ℛ
inheritance ≻ result ≙ result ∪ around inheritance

flowInherit :: Inheritable t ⇒ t → (ℛ → [ℛ]) → ℛ → [ℛ]
flowInherit t f r ≙ (map (inh ≻) ∘ f ∘ through) inh
  where
  inh ≙ t ≺ r
```

Finally, SNet networks are represented as an algebraic data type. The ADT is parametric, so that different stages of a compiler or run-time system can do structure preserving rewrites.

```
data Net
  a -- annotation, i.e. context
  b -- box function representation
  f -- snet filter representation
  p -- snet pattern representation
 ≙ Box      {ctxt :: a, n :: 𝔑, b :: 𝔅, f::b}
 | Filter   {ctxt :: a, n :: 𝔑, 𝓐 :: f }
 | Sync     {ctxt :: a, n :: 𝔑, 𝓟 :: [p]}
 | C_Seq    {ctxt :: a, n :: 𝔑, L :: [Net a b f p]}
 | C_Par    {ctxt :: a, n :: 𝔑, L :: [Net a b f p], δ :: Bool}
 | C_Split {ctxt :: a, n :: 𝔑, body :: Net a b f p, δ :: Bool, sel  :: ℒ}
 | C_Star   {ctxt :: a, n :: 𝔑, body :: Net a b f p, δ :: Bool, γ :: [p]}

netType :: Net a b f p → 𝔑
netType ≙ n
```

B.2 Expressions and patterns

```
{-# LANGUAGE GADTs, KindSignatures, MultiParamTypeClasses #-}
module Language.SNet.Evaluation where

import Language.SNet.Syntax
import qualified Data.IntMap as IM
```

Any evaluable type must be annotated with the type of the result of annotation. In other words, evaluating an instance of `a t` will result in an instance of `t`. Evaluation

takes a record as context. Names in expressions, types, etc. refer to the names as they occur in the context record[1].

```
class Evaluable a where
  eval :: ℛ → a t → t
```


Both filters and patterns have evaluable expressions. Filters can copy fields containing box language values and can perform common integer arithmatic on (binding) tags. Patterns match types *and* allow value dependencies on tag values, again including arithamtic. Their common expression language is represented with the ADT $\mathfrak{E}$.

```
data 𝔈 a where
  ℰ_ℒ   :: ℒ → 𝔈 𝒯
  ℰ_𝒯   :: 𝒯 → 𝔈 𝒯
  ℰ_×   ::[𝔈 𝒯] → 𝔈 𝒯
  ℰ_÷   ::[𝔈 𝒯] → 𝔈 𝒯
  ℰ_mod ::[𝔈 𝒯] → 𝔈 𝒯
  ℰ_+   ::[𝔈 𝒯] → 𝔈 𝒯
  ℰ_-   ::[𝔈 𝒯] → 𝔈 𝒯
  ℰ_min ::[𝔈 𝒯] → 𝔈 𝒯
  ℰ_max ::[𝔈 𝒯] → 𝔈 𝒯
  ℰ_|.| :: 𝔈 𝒯 → 𝔈 𝒯
  ℰ_¬   :: 𝔈 Bool → 𝔈 Bool
  ℰ_∧   ::[𝔈 Bool] → 𝔈 Bool
  ℰ_∨   ::[𝔈 Bool] → 𝔈 Bool
  ℰ_?   :: 𝔈 Bool → 𝔈 a → 𝔈 a → 𝔈 a
  (≒) :: Eq  a ⇒ 𝔈 a → 𝔈 a → 𝔈 Bool
  (≠) :: Eq  a ⇒ 𝔈 a → 𝔈 a → 𝔈 Bool
  (<) :: Ord a ⇒ 𝔈 a → 𝔈 a → 𝔈 Bool
  (≤) :: Ord a ⇒ 𝔈 a → 𝔈 a → 𝔈 Bool
  (>) :: Ord a ⇒ 𝔈 a → 𝔈 a → 𝔈 Bool
  (≥) :: Ord a ⇒ 𝔈 a → 𝔈 a → 𝔈 Bool
```

The evaluation of $\mathfrak{E}$ is rather unsurprising.

```
instance Evaluable 𝔈 where
  eval rec (ℰ_ℒ   n ) ≙ either (error "Non-int␣stored␣in␣tag") id (rec!n)
  eval rec (ℰ_𝒯   i ) ≙ i
  eval rec (ℰ_×   is) ≙ foldl1 (*) (map (eval rec) is)
  eval rec (ℰ_÷   is) ≙ foldl1 div (map (eval rec) is)
  eval rec (ℰ_mod is) ≙ foldl1 mod (map (eval rec) is)
  eval rec (ℰ_+   is) ≙ foldl1 (+) (map (eval rec) is)
  eval rec (ℰ_-   is) ≙ foldl1 (-) (map (eval rec) is)
  eval rec (ℰ_∧   bs) ≙ foldr1 (∧) (map (eval rec) bs)
  eval rec (ℰ_∨   bs) ≙ foldr1 (∨) (map (eval rec) bs)
  eval rec (ℰ_min is) ≙ minimum    (map (eval rec) is)
```

[1] In this type, `a t` is independent of `blv`. Most evaluable types are independent of `blv`. Only filter actions require this parameter. For brevity, the `blv` parameter is omitted where not explicitly relevant.

```
eval rec (ℰmax is) ≙ maximum    (map (eval rec) is)
eval rec (ℰ|.|  i ) ≙ abs (eval rec i)
eval rec (ℰ¬  b ) ≙ not (eval rec b)
eval rec (ℰ? i t e) | eval rec i ≙ eval rec t
                    | otherwise  ≙ eval rec e
eval rec ( l ≒ r ) ≙ eval rec l = eval rec r
eval rec ( l ≠ r ) ≙ eval rec l ≠ eval rec r
eval rec ( l ≺ r ) ≙ eval rec l < eval rec r
eval rec ( l ≼ r ) ≙ eval rec l ≤ eval rec r
eval rec ( l ≻ r ) ≙ eval rec l > eval rec r
eval rec ( l ≽ r ) ≙ eval rec l ≥ eval rec r
```

Patterns are used in synchrocells and as terminators for the star. All patterns evaluate to `Maybe ℜ`; if the pattern matches, the type of the matching part is returned. 'Plain' patterns ($\mathfrak{P}_P$) match a type. Guard patterns ($\mathfrak{P}_G$) match a type and evaluate an expression.

```
data 𝔓 a where
  𝔓P :: ℜ            → 𝔓 (Maybe ℜ)
  𝔓G :: ℜ → 𝔈 Bool → 𝔓 (Maybe ℜ)

instance Evaluable 𝔓 where
  eval rec ( 𝔓P t   ) | typeOf rec ≼ t ≙ Just t
                      | otherwise      ≙ Nothing
  eval rec ( 𝔓G t g ) | typeOf rec ≼ t ∧ eval rec g ≙ Just t
                      | otherwise                   ≙ Nothing
```

Filters consist of hierarchical structures of filter actions ($\mathfrak{A}$). The simplest actions are field actions ($\mathfrak{A}_f$) that can only copy fields. They are specified by two labels, the first is the label in the input record (source) and the second is the label in the output record (destination). The result of a field action is a single record entry, i.e. a combination of label and content. Tag actions ($\mathfrak{A}_t$) can contain expressions in $\mathfrak{E}$. The result of a tag action is a single record entry (label and content). A record action ($\mathfrak{A}_R$) consists of a sequence of field and/or tag actions. A 'plain' action consists of a sequence of record actions. Finally, a guard action evaluates a boolean expression, to decide which alternative action is taken.

```
data 𝔄 a where
  𝔄f :: ℒ →  ℒ  → 𝔄 (ℒ,𝒞)
  𝔄t :: ℒ → 𝔈 𝒯 → 𝔄 (ℒ,𝒞)
  𝔄R :: [𝔄 (ℒ,𝒞)] → 𝔄 ℛ
  𝔄P :: [𝔄 ℛ] → 𝔄 [ℛ]
  𝔄G :: 𝔈 Bool → 𝔄 a → 𝔄 a → 𝔄 a
```

The evaluation of actions is rather straightforward.

```
instance Evaluable 𝔄 where
  eval r (𝔄f d s  ) ≙ (d,r!s)
  eval r (𝔄t d s  ) ≙ (d,Right (eval r s))
```

```
eval r (𝔄R  as  ) ≙ IM.fromList (map (λa → eval r a) as)
eval r (𝔄P  as  ) ≙ map (λa → eval r a) as
eval r (𝔄G i t e) ≙ eval r (if eval r i then t else e)
```


B.3 Network indices

```
module Language.SNet.Index where
```

A network index must uniquely identify a network in a network expression. Network indices are made up out of *network index elements*. There are three kinds of network index elements to indicate networks: sequential (`Seq`), iteration (`Iter`) and alternative (`Alt`). Iteration network index elements are used to indicate in what unfolding of a serial replication a network sits. Alternative network index elements are used to indicate a choice for a branch in a split or parallel composition. There are to other kinds of network index elements, `Split` and `Merge`, that refer to the splitter and the merger of a parallel composition, respectively.

```
data NetIdxEl
  ≙ Seq  { val :: ℕ }
  | Iter { val :: ℕ }
  | Split
  | Alt  { val :: ℕ }
  | Merge
  deriving Eq
```

An network index (Ψ) is a list of network index elements. The list is organised 'deepest first', i.e. the first element in the list is the index of the innermost (or 'least significant') network structure. However, because the intuition from lexicographical order is to have the outermost (or 'most significant') network index element at the head of the list, we define an 'append' operator ($\oplus$) to add a network index element to the 'deep-end' of the network index.

```
type Ψ ≙ [NetIdxEl]

(⊕) :: Ψ → NetIdxEl → Ψ
(⊕) ≙ flip (:)
```

As an example, consider the network expression $A \mathbin{..} (B \mathbin{..} C)^{\star}_{\mathsf{F},\gamma} \parallel_{\mathsf{F}} D$. The index of network A is `[Seq 0]`. The index of network B is (for iteration i of the star B is in) `[Seq 0, Iter i, Alt 0, Seq 1]`.

Indices are partially ordered, so we can define an infimum. Since the most significant index element is the last element in the list, we reverse indices, recursively determine the infimum and reverse the result back to the proper order.

```
inf :: Ψ → Ψ → Ψ
inf l r ≙ reverse $ inf' (reverse l) (reverse r)
  where
```

```
inf' (l:ls) (r:rs) | l = r ≙ l : inf' ls rs
inf' ls•(Seq l:_) rs•(Seq r:_)
  | l < r ≙ ls
  | l > r ≙ rs
inf' ls•(Iter l:_) rs•(Iter r:_)
  | l < r ≙ ls
  | l > r ≙ rs
inf' [Split] (Alt _:_) ≙ [Split]
inf' (Alt _:_) [Split] ≙ [Split]
inf' [Merge] rs ≙ rs
inf' ls [Merge] ≙ ls
inf' _ _ ≙ []
```

Literate programming substitutions

C.1 Basic Haskell syntax

≙	=
•	@
←	<-
λ	\
→	->
⇒	=>
∘	.
¬	not
∧	&&
∨	||
≤	<=
=	==
≠	/=
≥	>=
ℕ	NAT
ℤ	INT
⊆	`isSubsetOf`
∈	`member`
∪	`union`
∃	`isSuffixOf`

C.2 Indices and oracles

ψ, Ψ	`index, Index`
ω, Ω	`oracle, Oracle`
$\upharpoonright$	`|>>`
$\oplus$	`+++`

C.3 SNet types & values and their operators

$\mathcal{L}$	`Label`
$\mathfrak{R}$	`RecType`
$\wp\mathcal{L}$	`LabelSet`
$\mathcal{T}$	`TagValue`
$\mathcal{C}$	`(Content blv)`
$\mathcal{R}$	`(Record blv)`
$\mathfrak{b}, \mathfrak{B}$	`boxtype, BoxType`
$\mathfrak{f}, \mathfrak{F}$	`funtype, FunType`
$\mathfrak{n}, \mathfrak{N}$	`fultype, FulType`
$f, \mathcal{F}$	`boxfunc, (BoxFunc blv)`
$\rightarrowtail$	`:->`
$\mapsto$	`:|>`
$\preccurlyeq$	`<:=`
$\prec$	`-|->`
$\succ$	`>-|-`
$\mathcal{I}$	`Inh`
$\mathfrak{I}$	`(Inherit blv)`

C.4 Types for program representation

C.4.1 Hiding the blv type parameter

`a`	`evaluable` and `evaluable blv`

C.4.2 Expressions

$\mathfrak{E}$	`Expr` and `(Expr blv)`
$\mathcal{E}_{\mathcal{L}}$	`ExprRef`
$\mathcal{E}_{\mathcal{T}}$	`ExprInt`
$\mathcal{E}_{\times}$	`ExprMul`
$\mathcal{E}_{\div}$	`ExprDiv`
$\mathcal{E}_{\text{mod}}$	`ExprMod`
$\mathcal{E}_{+}$	`ExprAdd`
$\mathcal{E}_{-}$	`ExprSub`
$\mathcal{E}_{\text{min}}$	`ExprMin`
$\mathcal{E}_{\text{max}}$	`ExprMax`
$\mathcal{E}_{\lvert.\rvert}$	`ExprAbs`
$\mathcal{E}_{\neg}$	`ExprNot`
$\mathcal{E}_{\wedge}$	`ExprAnd`
$\mathcal{E}_{\vee}$	`ExprOr`
$\mathcal{E}_{?}$	`ExprITE`
$\doteq$	`:==:`
$\neq$	`:!=:`
$\lessdot$	`:<:`
$\leqslant$	`:<=:`
$\gtrdot$	`:>:`
$\geqslant$	`:>=:`

C.4.3 Patterns

$\mathfrak{P}$	`Pattern` and `(Pattern blv)`
$\mathfrak{P}_G$	`GuardPattern`
$\mathfrak{P}_P$	`PlainPattern`

C.4.4 Actions

$\mathfrak{A}$	`Action` and `(Action blv)`
$\mathfrak{A}_R$	`RecAction`
$\mathfrak{A}_t$	`TagAction`
$\mathfrak{A}_f$	`FieldAction`
$\mathfrak{A}_G$	`GuardAction`
$\mathfrak{A}_P$	`PlainAction`

C.5 Semantics

C.5.1 Hiding the blv type parameter

`Router`	`(Router blv)`
`Branch`	`(Branch blv)`
`SyncState`	`(SyncState blv)`

C.5.2 Data types

$\mathfrak{s}$	`(Substream blv)`
$\mathfrak{S}$	`(Stream blv)`
$\widetilde{\mathfrak{S}}$	`(RoutedStream blv)`
$\mathcal{N}$	`(SemNet blv)`

C.5.3 Networks and semantics

δ	`delta`
γ	`terminator`
N	body and `subNet`
$[\![.]\!]$	`semantics`
$[\![N]\!]$	`semantics(body)` and `semantics(subNet)`
$\overline{[\![.]\!]}$	`semantics'`
$\overline{[\![N]\!]}$	`semantics'(subNet)`
$[\![\mathbf{box}(\mathfrak{b}, f)]\!]$	`semantics(Box {..})`
$[\![\mathbf{filter}(\mathfrak{f}, \mathcal{A})]\!]$	`semantics(Filter {..})`
$[\![\mathbf{sync}(\mathcal{P})]\!]$	`semantics(Sync {..})`
$\left[\!\left[\bullet\!\bullet_{N \leftarrow L} N \right]\!\right]$	`semantics(C_Seq {..})`
$[\![N^{\star}_{\delta,\gamma}]\!]$	`semantics(C_Star {..})`
$[\![N \,!_{\delta}\, t]\!]$	`semantics(C_Split {..})`
$\left[\!\left[\Vert_{\delta, N \leftarrow L} N \right]\!\right]$	`semantics(C_Par {..})`

Acronyms

ALU	Arithmetic and Logic Unit.
AP	Assignment Problem.
ASIC	Application Specific Integrated Circuit.
CMOS	Complementary Metal-Oxide-Semiconductor.
CRISP	Cutting edge Reconfigurable ICs for Stream Processing.
DDR	Double Data Rate.
DFG	dataflow graph.
DMA	Direct Memory Access.
DSP	Digital Signal Processing.
ELR	Early Latency Response.
FF	First Fit.
FFC	First Fit with Clustering.
FIFO	First-In First-Out.
FPGA	Field Programmable Gate Array.
FPM	Fast Page Mode.
GAP	Generalized Assignment Problem.
GPD	General Purpose Device.
GPP	General Purpose Processor.
HPC	High Performance Computing.
HwE	Hardware Element.
IC	Integrated Circuit.
ILP	Integer Linear Program.
IP	Intellectual Property.
LLR	Late Latency Response.
MIMO	Multiple Input Multiple Output.
MMAC	Million Multiply Accumulate.
MMKP	Multi-dimensional Multi-choice Knapsack Problem.
MP	Manager Processor.
MPSoC	Multi-Processor System-on-Chip.

NMOS	N-type Metal-Oxide-Semiconductor.
NoC	Network-on-Chip.
OS	Operating System.
PCB	Printed Circuit Board.
PE	Processing Element.
ProcFS	Process File System.
QoS	Quality of Service.
RFD	Reconfigurable Device.
RSRM	Run-time Spatial Resource Management.
SCC	Single-chip Cloud Computer.
SDF	Synchronous Data Flow.
SIMD	Single Instruction Multiple Data.
SiP	System in Package.
SISO	Single Input Single Output.
SOI	Silicon-On-Insulator.
TDMA	Time Division Multiple Access.
UCS	Uniform Cost Search.
VAP	Vector Assignment Problem.
VT	Virtual Tile.

Bibliography

[1] IBM ILOG CPLEX. URL http://www-01.ibm.com/software/integration/optimization/cplex/.

[2] Haskell :: Functional programming with types. URL http://en.wikibooks.org/wiki/Haskell.

[3] *Low-Power Domain-Specific Processors for Digital Signal Processing*. PhD thesis, University of California, Berkeley, 2001.

[4] *Coarse-Grained Reconfigurable Processors – Flexibility meets Efficiency*. PhD thesis, University of Twente, Enschede, The Netherlands, sep 2004.

[5] Apple. Mac OS X ABI mach-O file format reference on-line developer documentation, February 2009. URL http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/Mach-O_File_Format.pdf.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[7] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The fortran automatic coding system. In *IRE-AIEE-ACM '57 (Western): Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198, New York, NY, USA, 1957. ACM. doi: http://doi.acm.org/10.1145/1455567.1455599.

[8] K. Baynes, C. Collins, E. Fiterman, Brinda Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The performance and energy consumption of embedded real-time operating systems. *Computers, IEEE Transactions on*, 52(11):1454–1469, November 2003. ISSN 0018-9340. doi: 10.1109/TC.2003.1244943.

[9] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[10] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78, 2002. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.976921.

[11] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2): 87–152, November 1992.

[12] Tjerk Bijlsma,, Marco Bekooij,, Pierre Jansen,, and Gerard J. M. Smit,. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *SCOPES '08: Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 33–42, 2008.

[13] J Binder. Safety-critical software for aerospace systems. *Aerospace America*, pages 26–27, August 2004.

[14] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004. ISBN 0521833783. doi: 10.2277/0521833787.

[15] Timon ter Braak and Philip Kaj Ferdinand Hölzenspies. Kairos osrm, January 2010. URL `http://www.kairos-osrm.com/`.

[16] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*.

[17] Haoxan Cai, Susan Eisenbach, Alex Shafarenko, and Clemens Grelck. Extending the S-Net Type System. In *Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'07), Paris, France*, 2007.

[18] Ewerson Carvalho, Ney Calazans, and Fernando Moraes. Heuristics for dynamic task mapping in noC-based heterogeneous MPSocs. In *RSP '07: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, pages 34–40, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2834-1. doi: http://dx.doi.org/10.1109/RSP.2007.26.

[19] Chen-Ling Chou and Radu Marculescu. Incremental run-time application mapping for homogeneous nocs with multiple voltage levels. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 161–166, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-824-4. doi: http://doi.acm.org/10.1145/1289816.1289857.

[20] Reuven Cohen, Liran Katzir, and Danny Raz. An efficient approximation for the generalized assignment problem. *Inf. Process. Lett.*, 100(4):162–166, 2006. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/j.ipl.2006.06.003.

[21] Tilera Corporation. Tile64™ processor product brief. Corporate product brief, 2008. URL `http://www.tilera.com/pdf/ProBrief_Tile64_Web.pdf`.

[22] William James Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 0-12-200751-4.

[23] William James Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. Stream processors: Progammability and efficiency. *Queue*, 2(1):52–62, 2004. doi: 10.1145/984458.984486.

[24] Giovanni de Micheli and Luca Benini. Networks on chip: A new paradigm for systems on chip design. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 418, Washington, DC, USA, 2002. IEEE Computer Society.

[25] Rina Dechter, and Judea Pearl,. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 32(3):505–536, 1985. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/3828.3830.

[26] Advanced Micro Devices. Live migration with ADM-V extended migration technology. White paper, March 2008. URL `http://developer.amd.com/assets/LiveVirtualMachineMigrationonAMDprocessors.pdf`.

[27] Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8442-9.

[28] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. ISSN 0029-599X. doi: 10.1007/BF01386390. URL `http://dx.doi.org/10.1007/BF01386390`.

[29] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. doi: 10.1007/BF01386390. URL `http://dx.doi.org/10.1007/BF01386390`.

[30] *Broadband Radio Access Networks (BRAN); HiperLAN type 2; Physical (PHY) layer, ETSI TS 101 475 v1.2.2 (2001-02), 2001.* ETSI.

[31] Mohammad Abdullah Al Faruque, Rudolf Krist, and Jörg Henkel. ADAM: run-time agent-based distributed application mapping for on-chip communication. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 760–765, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6. doi: http://doi.acm.org/10.1145/1391469.1391664.

[32] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367766.368168.

[33] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput analysis of synchronous data flow graphs. *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 25–36, June 2006. ISSN 1550-4808. doi: 10.1109/ACSD.2006.33.

[34] A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen. Latency minimization for synchronous data flow graphs. *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 189–196, August 2007. doi: 10.1109/DSD.2007.4341468.

[35] Kees Goossens, John Dielissen, and Andrei Radulescu. æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, 2005. ISSN 0740-7475. doi: http://dx.doi.org/10.1109/MDT.2005.99.

[36] C. Grelck and F. Penczek. Implementation Architecture and Multithreaded Runtime System of S-Net. In S.B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers*, Lecture Notes in Computer Science. Springer-Verlag, 2009. to appear.

[37] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[38] Clemens Grelck, Jukka Julku, and Frank Penczek. Distributed S-Net. In M. Morazan, editor, *Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA*. Seton Hall University, 2009.

[39] Venkatesan Guruswami, Sanjeev Khanna, Rajmohan Rajaraman, Bruce Shepherd, and Mihalis Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. In *STOC '99: Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 19–28, New York, NY, USA, 1999. ACM. ISBN 1-58113-067-8. doi: http://doi.acm.org/10.1145/301250.301262.

[40] A. Hansson and K. Goossens. Trade-offs in the configuration of a network on chip for multiple use-cases. *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pages 233–242, May 2007. doi: 10.1109/NOCS.2007.45.

[41] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2005. ACM. ISBN 1-59593-161-9. doi: http://doi.acm.org/10.1145/1084834.1084857.

[42] Andreas Hansson, Maarten Wiggers, Arno Moonen, Kees Goossens, and Marco Bekooij. Applying dataflow analysis to dimension buffers for guaranteed performance in networks on chip. In *NOCS '08: Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 211–212, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3098-7.

[43] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):1–24, 2009. ISSN 1084-4309. doi: http://doi.acm.org/10.1145/1455229.1455231.

[44] Andreas Hansson, Maarten Wiggers, Arno Moonen, Kees Goossens, and Marco Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET Computers & Digital Techniques*, 2009.

[45] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2): 100–107, July 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

[46] Gernot Heiser. The role of virtualization in embedded systems. In *1st Workshop on Isolation and Integration in Embedded Systems*, pages 11–16, Glasgow, UK, Apr 2008. ACM SIGOPS. doi: 10.1145/1435458.

[47] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of period and sporadic tasks. *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139, December 1991. doi: 10.1109/REAL.1991.160366.

[48] Werner B. Joerg. A subclass of petri nets as design abstraction for parallel architectures. *SIGARCH Comput. Archit. News*, 18(4):67–77, 1990. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/121973.121982.

[49] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[50] G Kahn. The semantics of a simple language for parallel programming. In L Rosenfeld, editor, *Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden*, pages 471–475. North-Holland, 1974.

[51] D. I. Kang,, J. Suh,, O. McMahon,, and S. Crago,. Preliminary study towards intelligent run-time resource management techniques for large multi-core architectures. Technical report, University of Southern California – Information Sciences Institute, September 2007. High-Performance Computing Workshop.

[52] N. K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. PhD thesis, University of Twente, Enschede, January 2007.

[53] N. K. Kavaldjiev, G. J. M. Smit, and P. G. Jansen. A virtual channel router for on-chip networks. In *IEEE Int. SOC Conf., Santa Clara, California*, pages 289–293, Los Alamitos, California, September 2004. IEEE Computer Society. ISBN 0-7803-8445-8.

[54] Nikolay Kavaldjiev, Gerard J. M. Smit, Pascal T. Wolkotte, and Pierre G. Jansen. Providing qoS guarantees in a noC by virtual channel reservation. In *ARC*, pages 299–310, 2006.

[55] Jong-Kook Kim, Sameer Shivle, Howard Jay Siegel, Anthony A. Maciejewski, Tracy D. Braun, Myron Schneider, Sonja Tideman, Ramakrishna Chitta, Raheleh B. Dilmaghani, Rohit Joshi, Aditya Kaul, Ashish Sharma, Siddhartha Sripada, Praveen Vangari, and Siva Sankar Yellampalli. Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 98.1, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1926-1.

[56] Reiner Kolisch, Arno Sprecher, and Andreas Drexl. *Management Science*, 41(10):1693–1703, 1995. ISSN 00251909. URL `http://www.jstor.org/stable/2632747`.

[57] Oscar J. Kuiken,, Xiao Zhang,, and Hans G. Kerkhoff,. Built-in self-diagnostics for a noC-based reconfigurable IC for dependable beamforming applications. In *DFT '08: Proceedings of the 2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 45–53, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3365-0. doi: http://dx.doi.org/10.1109/DFT.2008.24.

[58] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun. Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip. In *ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time*

Multimedia, pages 33–38, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7803-9783-5. doi: http://dx.doi.org/10.1109/ESTMED.2006.321271.

[59] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

[60] Harold Lawson and Howard Bromberg. The world's first cobol compilers. transcript at `http://www.computerhistory.org/events/lectures/cobol_06121997/`, 1997.

[61] E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. In *Proceedings of the IEEE*, volume 75(9), pages 1235 – 1245, September 1987.

[62] Edward A. Lee and Thomas M. Parks. Dataflow process networks. pages 59–85, 2002.

[63] Anany Levitin,. Do we teach the right algorithm design techniques? *SIGCSE Bull.*, 31 (1):179–183, 1999. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/384266.299747.

[64] Keqin Li and Kam Hoi Cheng. A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. In *CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation*, pages 22–27, New York, NY, USA, 1990. ACM. ISBN 0-89791-348-5. doi: http://doi.acm.org/10.1145/100348.100352.

[65] César Marcon, André Borin, Altamiro Susin, Luigi Carro, and Flávio Wagner. Time and energy efficient mapping of embedded applications onto nocs. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 33–38, New York, NY, USA, 2005. ACM. ISBN 0-7803-8737-6. doi: http://doi.acm.org/10.1145/1120725.1120738.

[66] Théodore Marescaux and Henk Corporaal. Introducing the supergt network-on-chip: Supergt qos: more than just gt. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 116–121, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: http://doi.acm.org/10.1145/1278480.1278510.

[67] Silvano Martello, and Paolo Toth,. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990. ISBN 0-471-92420-2.

[68] Orlando Moreira, Jacob Jan-David Mol, and Marco Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1557–1564, New York, NY, USA, 2007. ACM. ISBN 1-59593-480-4. doi: http://doi.acm.org/10.1145/1244002.1244335.

[69] Orlando M. Moreira, and Marco J. G. Bekooij,. Self-timed scheduling analysis for real-time applications. In *EURASIP Journal on Advances in Signal Processing*, volume 2007, pages 24–37. Hindawi Publishing Corporation, April 2007. doi: doi:10.1155/2007/83710.

[70] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: Toward composable multimedia MP-soC design. *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 574–579, June 2008. ISSN 0738-100X.

[71] V. Nollet, P. Avasare, J-Y. Mignolet, and D. Verkest. Low cost task migration initiation in a heterogeneous MP-soC. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 252–253, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2288-2. doi: http://dx.doi.org/10.1109/DATE.2005.201.

[72] V. Nollet, T. Marescaux, P. Avasare, and J-Y. Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 234–239, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2288-2. doi: http://dx.doi.org/10.1109/DATE.2005.91.

[73] Zeev Nutov, Israel Beniaminy, and Raphael Yuster. A (1-1/ε)-approximation algorithm for the generalized assignment problem. *Oper. Res. Lett.*, 34(3):283–288, 2006.

[74] Members of the FreeBSD Documentation Project. *FreeBSD Handbook*. Dancing Goat Press, 2 edition, November 2001. ISBN 1-57176-303-1.

[75] T. Ojanpera and R. Prasad. An overview of air interface multiple access for imt-2000/umts. *IEEE Communications Magazine*, 36(9):82–95, September 1998.

[76] Frank Penczek. Design and Implementation of a Multithreaded Runtime System for the Stream Processing Language S-Net. Master's thesis, Institute of Software Technology and Programming Languages, University of Lübeck, Germany, 2007.

[77] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[78] Carl Adam Petri. *Concurrency as a Basis of Systems Thinking.* St. Augustin: Gesellschaft für Mathematik und Datenverarbeitung Bonn, Interner Bericht ISF-78–06, September 1978.

[79] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. 2003.

[80] Recore Systems BV. CRISP - cutting edge reconfigurable ICs for stream processing, February 2008. URL `http://www.crisp-project.eu`. FP7-ICT-215881.

[81] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. ISBN 0137903952. URL `http://portal.acm.org/citation.cfm?id=773294`.

[82] Intel Tera scale Computing Research Program. Single-chip cloud computer information page. URL `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`.

[83] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[84] Merrill I. Skolnik. *Introduction to Radar Systems*. McGraw-Hill, New York, NY, USA, third edition, 2001. ISBN 0-07-288138-0. URL `http://www.mhprofessional.com/product.php?isbn=0072881380`.

[85] G. J. M. Smit, A. B. J. Kokkeler, P. T. Wolkotte, and M. D. van de Burgwal. Multi-core architectures and streaming applications. In I. Mandoiu and A. Kennings, editors, *Proceedings of the Tenth International Workshop on System-Level Interconnect Prediction (SLIP 2008), Newcastle, UK*, pages 35–42, New York, NY, USA, April 2008. ACM.

[86] SNet. SNet declarative coordination language, 2008. URL `http://www.snet-home.org/`.

[87] K. Srinivasan and K. S. Chatha. A technique for low energy mapping and routing in network-on-chip architectures. *Low Power miscs and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, pages 387–392, August 2005.

[88] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.7053.

[89] Ralph Steuer. *Multiple Criteria Optimization: Theory, Computation, and Application*. Krieger Publishing Company, January 1986.

[90] Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6 (5):611–624, 1998. ISSN 1063-6692. doi: 10.1109/90.731196.

[91] Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Trans. Netw.*, 6(5):611–624, 1998. ISSN 1063-6692. doi: http://dx.doi.org/10.1109/90.731196.

[92] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Programming Research Group Technical Monograph PRG-11, Oxford Univ. Computing Lab., 1974. Reprinted in *Higher-Order and Symbolic Computation*, vol. 13 (2000), pp. 135–152.

[93] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. URL `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[94] top500 project. The 34th top500 list, Nov 2009. URL `www.top500.org`.

[95] Franco Travostino, Paul Daspit, Leon Gommans, Chetan Jog, Cees de Laat, Joe Mambretti, Inder Monga, Bas van Oudenaarde, Satish Raghunath, and Phil Yonghui Wang. Seamless live migration of virtual machines over the man/wan. *Future Gener. Comput. Syst.*, 22(8):901–907, 2006. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2006.03.007.

[96] A. van Wijngaarden. Recursive definition of syntax and semantics. In *Formal Language Description Languages for Computer Programming. Proceedings of the IFIP Working Conference on Formal Language Description Languages.*, pages 13–24, Amsterdam, 1966. North-Holland Publishing Company.

[97] Sriram Vangal, James Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskote, and Nitin Borkar. An 80-tile 1.28 tflops network-on-chip in 65nm cmos. In *Proceedings of the International Solid State Circuits Conference*. IEEE, IEEE, February 2007.

[98] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007(1):19–19, 2007. ISSN 1687-3955. doi: http://dx.doi.org/10.1155/2007/75947.

[99] VMware. VMware VMontion and CPU compatibility. Information Guide, June 2008. URL `http://www.vmware.com/files/pdf/vmotion_info_guide.pdf`.

[100] W. Warner. Great moments in microprocessor history, the history of the micro from the vacuum tube to today's dual-core multithreaded madness, Dec 2004. URL `http://www.ibm.com/developerworks/library/pa-microhist.html`.

[101] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Computation of buffer capacities for throughput constrained and data dependent inter-task communication. In *Design Automation and Test in Europe, Munich*, pages 640–645, San Jose, CA, USA, March 2008. EDA Consortium.

[102] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS'08, St. Louis, MO, United States*, pages 183–194, Los Alamitos, April 2008. IEEE Computer Society Press.

[103] Maarten H. Wiggers,, Marco J. G. Bekooij,, and Gerard J. M. Smit,. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pages 11–22, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1269843.1269846.

[104] Maarten Hendrik Wiggers. *Aperiodic multiprocessor scheduling for real-time stream processing applications*. PhD thesis, University of Twente, Enschede, June 2009. URL `http://doc.utwente.nl/61568/`.

[105] P. T. Wolkotte, G. J. M. Smit, N. Kavaldjiev, J. E. Becker, and J. Becker. Energy model of networks-on-chip and a bus. *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pages 82–85, November 2005. doi: 10.1109/ISSOC.2005.1595650.

[106] Pascal T. Wolkotte,. *Exploration within the Network-on-Chip Paradigm*. PhD thesis, University of Twente, 2009.

[107] Ch. Ykman-Couvreur, V. Nollet, Fr. Catthoor, and H. Corporaal. Fast multi-dimension multi-choice knapsack heuristic for MP-soC run-time management. *System-on-Chip, 2006. International Symposium on*, pages 1–4, November 2006. doi: 10.1109/ISSOC.2006.321966.

[108] Ch. Ykman-Couvreur, V. Nollet, Th. Marescaux, E. Brockmeyer, Fr. Catthoor, and H. Corporaal. Design-time application mapping and platform exploration for MP-soC customised run-time management. *Computers & Digital Techniques, IET*, 1(2): 120–128, March 2007. ISSN 1751-8601. doi: 10.1049/iet-cdt:20060031.

[109] Kun Zhang, and Santosh Pande,. Minimizing downtime in seamless migrations of mobile applications. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 12–21, New York, NY, USA, 2006. ACM. ISBN 1-59593-362-X. doi: http://doi.acm.org/10.1145/1134650.1134654.

[110] X. Zhang and H. G. Kerkhoff. Design of a highly dependable beamforming chip. In X. Zhang, editor, *Proceedings of Euromicro on Digital System Design (DSD09), Patras, Greece*, pages 729–735, Los Alamitos, CA, USA, August 2009. IEEE Computer Society Press.

List of Publications

[PhH:1] C. Grelck, Shafarenko, A. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, Scholz, S.B., and A. Shafarenko. S-Net Language Report 1.0. Technical Report 487, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2009.

[PhH:2] P. K. F. Hölzenspies, J. Kuper, G. J. M. Smit, and J. L. Hurink. Demonstration of run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc). In B. R. H. M. Haverkort, J. P. Katoen, and L. Thiele, editors, *Dagstuhl Seminar Proceedings 07101, Dagstuhl Wadern, Germany*, volume 07101, Dagstuhl, Germany, October 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI).

[PhH:3] P. K. F. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. M. Smit. Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc). In *Proceedings of the Eleventh Conference on Design, Automation and Test in Europe, DATE08, Munich, Germany*, pages 212–217. European Design and Automation Association, March 2008.

[PhH:4] Philip Hölzenspies, Timon ter Braak, Jan Kuper, Gerard Smit, and Johann Hurink. Run-time spatial mapping of streaming applications to heterogeneous multi-processor systems. *International Journal of Parallel Programming*. doi: 10.1007/s10766-009-0120-y.

[PhH:5] P.K.F. Hölzenspies, G.J.M. Smit, and J. Kuper. Mapping streaming applications on a reconfigurable mpsoc platform at run-time. In *Proceedings of the International Symposium on System-on-Chip (SoC 2007)* [Best paper award], pages 74–77, Tampere, Finland, November 2007. IEEE Circuits and Systems Society. ISBN 1-4244-1367-2.

[PhH:6] A. B. J. Kokkeler, G. K. Rauwerda, P. T. Wolkotte, Q. Zhang, P. K. F. Hölzenspies, and G. J. M. Smit. Reconfigurable baseband processing for wireless communications. In M. Ibnkahla, editor, *Adaptive Signal Processing in Wireless Communications (Adaptation in Wireless Communications)*, pages 443–478. CRC Press, Boca Raton, Florida, US, 2009. ISBN 1-420-04601-2.

[PhH:7] G. J. M. Smit, A. B. J. Kokkeler, P. T. Wolkotte, P. K. F. Hölzenspies, M. D. van de Burgwal, and P. M. Heysters. The chameleon architecture for streaming dsp applications. *EURASIP Journal on Embedded Systems*, 2007:78082, 2007. ISSN 1687-3955.

[PhH:8] Timon D. ter Braak, Philip K.F. Hölzenspies, Jan Kuper, Johann L. Hurink, and Gerard J.M. Smit. Run-time spatial resource management for real-time applications on heterogeneous mpsocs. In *Proceedings of the Thirteenth Conference on Design, Automation and Test in Europe, DATE10, Dresden, Germany*. European Design and Automation Association, March 2010.

[PhH:9] Jeyarajan Thiyagalingam, Philip Hölzenspies, Sven-Bodo Scholz, and Alex Shafarenko. A Stream-Order Relaxed Execution Model forAsynchronous Stream Languages. In Sven-Bodo Scholz, editor, *Implementation and Application of Functional Languages, 20th international symposium, IFL'08, Hatfield, Hertfordshire, UK*, Technical Report 474, pages 316–329. University of Hertfordshire, England, UK, 2008.

[PhH:10] P. T. Wolkotte, P. K. F. Hölzenspies, and G. J. M. Smit. Using an fpga for fast bit accurate soc simulation. In *Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS'07) - 14th Reconfigurable Architecture Workshop (RAW 2007), Long Beach, CA, USA*, number 07TH8938, page 167, Piscataway, March 2007. IEEE Computer Society Press. ISBN 1-4244-0910-1.

[PhH:11] P. T. Wolkotte, P. K. F. Hölzenspies, and G. J. M. Smit. Fast, accurate and detailed noc simulations. In P. Kellenberger, editor, *Proceedings of the 1st ACM/IEEE International Symposium on Networks-on-Chip, Princeton, NJ, USA*, number P2773, pages 323–332, Los Alamitos, May 2007. IEEE Computer Society Press. ISBN 0-7695-2773-6.

[PhH:12] P. T. Wolkotte, J. H. Rutgers, P. K. F. Hölzenspies, M. Westmijze, R. Blumink, and G. J. M. Smit. An automated design-flow for fpga-based sequential simulation. In *Proceedings of the 19th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC), Veldhoven, The Netherlands*, number 2008/14935/STW, pages 126–132, Utrecht, November 2008. STW.