IMPROVING MODEL-BASED SOFTWARE SYNTHESIS

A Focus on Mathematical Structures

Dissertation

zur Erlangung des akademischen Grades
Doktor rerum naturalium (Dr.rer.nat.).

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Andrés Wilhelm Goens Jokisch
geboren am 23.09.1989 in San Salvador

Gutachter:
Prof. Dr.-Ing. Jeronimo Castrillon
Technische Universität Dresden

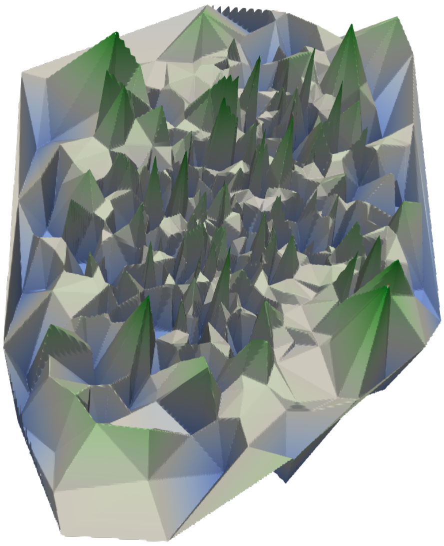
Dhr. prof. dr. Andy Pimentel
Universiteit van Amsterdam

Tag der Vertiedigung:
30.04.2021

TECHNISCHE
UNIVERSITÄT
DRESDEN

IMPROVING MODEL-BASED SOFTWARE SYNTHESIS

A Focus on Mathematical Structures



Andrés Wilhelm Goens Jokisch
May 2021 – 1.1

Dedicated to all who are unjustly opressed only for being born with a specific sex, race or species.

PREAMBLE

> All models are wrong, but some are useful
>
> George Box (Attributed)

Abstract

Computer hardware keeps increasing in complexity. Software design needs to keep up with this. The right models and abstractions empower developers to leverage the novelties of modern hardware. This thesis deals primarily with Models of Computation, as a basis for software design, in a family of methods called software synthesis.

We focus on Kahn Process Networks and dataflow applications as abstractions, both for programming and for deriving an efficient execution on heterogeneous multicores. The latter we accomplish by exploring the design space of possible mappings of computation and data to hardware resources. Mapping algorithms are not at the center of this thesis, however. Instead, we examine the mathematical structure of the mapping space, leveraging its inherent symmetries or geometric properties to improve mapping methods in general.

This thesis thoroughly explores the process of model-based design, aiming to go beyond the more established software synthesis on dataflow applications. We starting with the problem of assessing these methods through benchmarking, and go on to formally examine the general goals of benchmarks. In this context, we also consider the role modern machine learning methods play in benchmarking.

We explore different established semantics, stretching the limits of Kahn Process Networks. We also discuss novel models, like Reactors, which are designed to be a deterministic, adaptive model with time as a first-class citizen. By investigating abstractions and transformations in the Ohua language for implicit dataflow programming, we also focus on programmability.

The focus of the thesis is in the models and methods, but we evaluate them in diverse use-cases, generally centered around Cyber-Physical Systems. These include the 5G telecommunication standard, automotive and signal processing domains. We even go beyond embedded systems and discuss use-cases in GPU programming and microservice-based architectures.

Publications

Some contents of this thesis have been published previously, including ideas and some figures. The following are the publications cited in this thesis that I co-authored:

[Ode+14] Maximilian Odendahl, Andrés Goens, Rainer Leupers, Gerd Ascheid, Benjamin Ries, and Berthold Vöckingand Tomas Henriksson. “Optimized buffer allocation in multicore platforms.” In: *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association. 2014, p. 324.

[GC15] Andrés Goens and Jeronimo Castrillon. “Analysis of Process Traces for Mapping Dynamic KPN Applications to MPSoCs.” In: *System Level Design from HW/SW to Memory for Embedded Systems. IESS 2015. IFIP Advances in Information and Communication Technology, vol 523*. Ed. by Marcelo Götz, Gunar Schirner, Marco Aurélio Wehrmeister, Mohammad Abdullah Al Faruque, and Achim Rettberg. Foz do Iguaçu, Brazil: Springer International Publishing, Nov. 2015, pp. 116–127. ISBN: 978-3-319-90023-0. DOI: 10.1007/978-3-319-90023-0_10. URL: https://link.springer.com/chapter/10.1007%5C%2F978-3-319-90023-0_10.

[Ode+15] Maximilian Odendahl, Andrés Goens, Rainer Leupers, Gerd Ascheid, and Tomas Henriksson. “Buffer allocation based on-chip memory optimization for many-core platforms.” In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE. 2015, pp. 1119–1124.

[GCL16] Andrés Goens, Jeronimo Castrillon, and Maximilian Odendahland Rainer Leupers. “An Optimal Allocation of Memory Buffers for Complex Multicore Platforms.” In: *Journal of Systems Architecture* 66-67 (May 2016), pp. 69–83. DOI: 10.1016/j.sysarc.2016.05.002.

[Goe+16] Andrés Goens, Robert Khasanov, Jeronimo Castrillon, Simon Polstra, and Andy Pimentel. “Why Comparing System-level MPSoC Mapping Approaches is Difficult: a Case Study.” In: *Proceedings of the IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16)*. Ecole Centrale de Lyon, Lyon, France, Sept. 2016, pp. 281–288. DOI: 10.1109/MCSoC.2016.48. URL: https://cfaed.tu-dresden.de/files/user/jcastrillon/publications/1609_Goens_MCSoC.pdf.

[MGC16] Christian Menard, Andrés Goens, and Jeronimo Castrillon. “High-Level NoC Model for MPSoC Compilers.” In: *Proceedings of the IEEE Nordic Circuits and Systems Conference (NORCAS’16)*. NORCAS. Copenhagen, Denmark, Nov. 2016, pp. 1–6. DOI: 10.1109/NORCHIP.2016.7792876. URL: https://cfaed.tu-dresden.de/files/user/jcastrillon/publications/1611_Menard_NORCAS.pdf.

[Völ+16] Marcus Völp, Sascha Klüppelholz, Jeronimo Castrillon, Hermann Härtig, Nils Asmussen, Uwe Assmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, Andrés Goens, Sebastian Haas, Dirk Habich, Mattis Hasler, Immo Huismann, Tomas Karnagel, Sven Karol, Wolfgang Lehner, Linda Leuschner, Matthias Lieber, Siqi Ling, Steffen Märcker, Johannes Mey, Wolfgang Nagel, Benedikt Nöthen, Rafael Peñaloza, Michael Raitza, Jörg Stiller, Annett Ungethüm, and Axel Voigt. “The Orchestration Stack: The Impossible Task of Designing Software for Unknown Future Post-CMOS Hardware.” In: *Proceedings of the 1st International Workshop on Post-Moore’s Era Supercomputing (PMES), Co-located with The International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*. Salt Lake City, USA, Nov. 2016. URL: https://cfaed.tu-dresden.de/files/user/jcastrillon/publications/1611_Voelp_PMES.pdf.

[Goe+17] Andrés Goens, Robert Khasanov, Marcus Hähnel, Till Smejkal, Hermann Härtig, and Jeronimo Castrillon. “TETRiS: a Multi-Application Run-Time System for Predictable Execution of Static Mappings.” In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES’17)*. SCOPES ’17. Sankt Goar, Germany: ACM, June 2017, pp. 11–20. ISBN: 978-1-4503-5039-6. DOI: 10.1145/3078659.3078663. URL: http://doi.acm.org/10.1145/3078659.3078663.

[GSC17] Andrés Goens, Sergio Siccha, and Jeronimo Castrillon. “Symmetry in Software Synthesis.” In: *ACM Transactions on Architecture and Code Optimization (TACO)*, 14.2 (July 2017), 20:1–20:26. ISSN: 1544-3566. DOI: 10.1145/3095747. eprint: arXiv:1704.06623. URL: http://doi.acm.org/10.1145/3095747.

[Hem+17] Gerald Hempel, Andrés Goens, Josefine Asmus, Jeronimo Castrillon, and Ivo F. Sbalzarini. “Robust Mapping of Process Networks to Many-Core Systems Using Bio-Inspired Design Centering.” In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES ’17)*. SCOPES ’17. Sankt Goar, Germany: ACM, June 2017, pp. 21–30. ISBN: 978-1-4503-5039-6. DOI: 10.1145/3078659.3078667. URL: http://doi.acm.org/10.1145/3078659.3078667.

[Cas+18] Jeronimo Castrillon, Matthias Lieber, Sascha Klüppelholz, Marcus Völp, Nils Asmussen, Uwe Assmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, Andrés Goens, Sebastian Haas, Dirk Habich, Hermann Härtig, Mattis Hasler, Immo Huismann, Tomas Karnagel, Sven Karol, Akash Kumar, Wolfgang Lehner, Linda Leuschner, Siqi Ling, Steffen Märcker, Christian Menard, Johannes Mey, Wolfgang Nagel, Benedikt Nöthen, Rafael Peñaloza, Michael Raitza, Jörg Stiller, Annett Ungethüm, Axel Voigt, and Sascha Wunderlich. “A Hardware/Software Stack for Heterogeneous Systems.” In: *IEEE Transactions on Multi-Scale Computing Systems* 4.3 (July 2018), pp. 243–259. ISSN: 2332-7766. DOI: 10.1109/TMSCS.2017.2771750. URL: http://ieeexplore.ieee.org/document/8103042/.

[Ert+18] Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. "Compiling for Concise Code and Efficient I/O." In: *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. CC 2018. Vienna, Austria: ACM, Feb. 2018, pp. 104–115. DOI: `10.1145/3178372.3179505`. URL: `https://dl.acm.org/citation.cfm?id=3179505`.

[Goe+18] Andrés Goens, Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. "Level Graphs: Generating Benchmarks for Concurrency Optimizations in Compilers." In: *Proceedings of the 11th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG'2018), co-located with 13th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*. Manchester, United Kingdom, Jan. 2018. URL: `http://research.ac.upc.edu/multiprog/multiprog2018/papers/MULTIPROG-2018_Goens.pdf`.

[GMC18] Andrés Goens, Christian Menard, and Jeronimo Castrillon. "On the Representation of Mappings to Multicores." In: *Proceedings of the IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-18)*. Vietnam National University, Hanoi, Vietnam, Sept. 2018, pp. 184–191. DOI: `10.1109/MCSoC2018.2018.00039`. URL: `https://ieeexplore.ieee.org/document/8540232`.

[KGC18] Robert Khasanov, Andrés Goens, and Jeronimo Castrillon. "Implicit Data-Parallelism in Kahn Process Networks: Bridging the MacQueen Gap." In: *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM'18), co-located with 13th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*. PARMA-DITAM '18. Manchester, United Kingdom: ACM, Jan. 2018, pp. 20–25. ISBN: 978-1-4503-6444-7. DOI: `10.1145/3183767.3183790`. URL: `http://doi.acm.org/10.1145/3183767.3183790`.

[Ert+19a] Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. "Category-Theoretic Foundations of "STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism"." In: *CoRR* abs/1906.12098 (June 2019). arXiv: `1906.12098`. URL: `http://arxiv.org/abs/1906.12098`.

[Ert+19b] Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. "STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism." In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: ACM, Aug. 2019, pp. 146–161. ISBN: 978-1-4503-6813-1. DOI: `10.1145/3331545.3342600`. URL: `http://doi.acm.org/10.1145/3331545.3342600`.

[Goe+19] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. "A

Case Study on Machine Learning for Synthesizing Benchmarks." In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*. MAPL 2019. Phoenix, AZ, USA: ACM, June 2019, pp. 38–46. DOI: 10.1145/3315508.3329976. URL: http://doi.acm.org/10.1145/3315508.3329976.

[GMC19] Andrés Goens, Christian Menard, and Jeronimo Castrillon. "On Compact Mappings for Multicore Systems." In: *Proceedings of the IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*. Ed. by D. Pnevmatikatos, M. Pelcat, and M. Jung. Vol. 11733. IEEE. Pythagorion, Greece: Springer, Cham, July 2019, pp. 325–335. ISBN: 978-3-030-27561-7. DOI: 10.1007/978-3-030-27562-4_23. URL: https://link.springer.com/chapter/10.1007/978-3-030-27562-4_23.

[Loh+19] Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. "Actors Revisited for Time-Critical Systems." In: *Proceedings of the 56th annual Design Automation Conference*. DAC 2019. Las Vegas, NV, USA: ACM, June 2019, 4pp. DOI: 10.1145/3316781.3323469. URL: http://doi.acm.org/10.1145/3316781.3323469.

[BGC20] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. "ComPy-Learn: A Toolbox for Exploring Machine Learning Representations for Compilers." In: *2020 Forum for Specification and Design Languages (FDL)*. Kiel, Germany, Sept. 2020.

[Bra+20] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. "Compiler-Based Graph Representations for Deep Learning Models of Code." In: *Proceedings of the 29th ACM SIGPLAN International Conference on Compiler Construction (CC 2020)*. CC 2020. San Diego, CA, USA: Association for Computing Machinery, Feb. 2020, pp. 201–211. ISBN: 9781450371209. DOI: 10.1145/3377555.3377894. URL: https://doi.org/10.1145/3377555.3377894.

[Kha+20] Asif Ali Khan, Andrés Goens, Fazal Hameed, and Jeronimo Castrillon. "Generalized Data Placement Strategies for Racetrack Memories." In: *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)*. DATE '20. Grenoble, France: IEEE, Mar. 2020, pp. 1502–1507. ISBN: 978-3-9819263-4-7. DOI: 10.23919/DATE48585.2020.9116245. URL: https://ieeexplore.ieee.org/document/9116245.

[Loh+20c] Marten Lohstroh, Íñigo Íncer Romero, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. "Reactors: A Deterministic Model for Composable Reactive Systems." In: *Cyber Physical Systems. Model-Based Design – Proceedings of the 9th Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy 2019) and the Workshop on Embedded and Cyber-Physical Systems Education (WESE 2019)*. Ed. by Roger Chamberlain, Martin Edin Grimheden, and Walid Taha. New York City, NY, USA: Springer International Publishing, Feb. 2020, pp. 59–85. ISBN: 978-3-030-41131-2. DOI: 10.1007/978-3-030-41131-2_4. URL:

https://link.springer.com/chapter/10.1007/978-3-030-41131-2_4.

[Men+20] Christian Menard, Andrés Goens, Marten Lohstroh, and Jeronimo Castrillon. “Achieving Determinism in Adaptive AUTOSAR.” In: *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)*. DATE ’20. Grenoble, France: IEEE, Mar. 2020, pp. 822–827. ISBN: 978-3-9819263-4-7. DOI: 10.23919/DATE48585.2020.9116430. URL: https://ieeexplore.ieee.org/abstract/document/9116430.

[Wit+20] Robert Wittig, Andrés Goens, Christian Menard, Emil Matus, Gerhard P. Fettweis, and Jeronimo Castrillon. “Modem Design in the Era of 5G and Beyond: The Need for a Formal Approach.” In: *Proceedings of the 27th International Conference on Telecomunications (ICT)*. Bali, Indonesia, Oct. 2020.

[Men+21] Christian Menard, Andrés Goens, Gerald Hempel, Robert Khasanov, Julian Robledo, Felix Teweleitt, and Jeronimo Castrillon. “Mocasin – Rapid Prototyping of Rapid Prototyping Tools: A Framework for Exploring New Approaches in Mapping Software to Heterogeneous Multi-cores.” In: *Proceedings of the 13th RAPIDO Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, co-located with 16th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*. RAPIDO ’21. Budapest, Hungary: ACM, Jan. 2021.

[GNC] Andrés Goens, Timo Nicolai, and Jeronimo Castrillon. “mpsym: Improving Design-Space Exploration of Clustered Manycores with Arbitrary Topologies.” Manuscript submitted for publication.

Acknowledgments

First and foremost, I thank my advisor Jeronimo Castrillon. I consider him to have been both a mentor and a friend during the time I've spent working on this thesis. His advice shaped my research and this thesis would not exist without his guidance and help.

I also want to thank my current and former colleagues and co-authors at the chair for compiler construction: Justus Adam, Hasna Bouraoui, Alexander Brauckmann, Sebastian Ertel, Fazal Hameed, Gerald Hempel, Sven Karol, Asif Khan, Robert Khasanov, Nesrine Khouzami, Christian Menard, Norman Rink, Julian Robledo, Lars Schütze and Felix Wittwer. Thank you for creating a great environment to learn and work together, for countless discussions and insights, for your patience with my insistance on going to Zeltmensa and the great discussions that arose there, and for offering my comradery and friendship.

I want to thank everyone who worked with me as a student, helping me realize my research vision, from whom I've also learned a great deal, and some of whom have become colleagues in the meantime. Concretely, thank you, Alexander Brauckmann, Sebastian Krammer, Christian Menard, Timo Nicolai, Marcus Rossel, Alexander Thierfelder, Felix Teweleitt and Markus Walter.

Thanks to Silexica for letting me work with their product, which started as a spinoff of Multi-Processor System-on-Chip (MPSoC) Application Programming Studio (MAPS). Special thanks go to Luis Murillo for the patience of reading through all my papers related to Silexica and also being a source of inspiration in this collaboration. Mostly however, I want to thank Max Odendahl, for trusting in my abilities while knowing me only on a personal level, and introducing me to the field. Without him and Aufwärts Aachen, I would not be where I am today, thank you!

During my Ph.D I had the opportunity to visit Andy Pimentel at the University of Amsterdam, where I was warmly received by him, Simon Polstra and the rest of the group. Thank you for welcoming me and for a fruitful collaboration. I'd also like to thank the HiPEAC project for funding this visit through a collaboration grant. I also had the opportunity to visit Edward Lee at the University of Califonria at Berkeley. There, Matt Weber and Gil Lederman received me in their office, where I felt very welcome, like any other colleague. I want to thank both, as well as Marten Lohstroh, all of whom I had great discussions with, and who made my visit at Berkeley extremely fruitful. A special thank you also goes to Mary Stewart for helping sort out everything there, even to the point of making sure I had something to eat at the group lunches. Most of all, I would like to thank Edward Lee for accepting me to visit his group and taking the time to talk with me regularly. This visit was a pivotal point in my Ph.D. and I really appreciated everything and everyone. Outside the academic realm, I want to thank Giulia Leggett for making this visit extremely enriching also from a personal point of view. I also want to thank the German foreign exchange service DAAD and specifically the FIT Weltweit project, as well, the Center for Advancing Electronics Dresden (cfaed) cluster of excellence, for helping me finance this visit.

I also want to thank the rest of my co-authors. Colloaboration with them made this thesis possible. Thanks to Chris Cummins and Hugh Leather for being so open in our collaboration and for their hospitality in Edingburgh. To everyone in the cfead Orchestration path, for sharing a vision with me and constructive retreats. I also thank Marcus Hähnel and Till Smejkal for a very successful collaboration what started the TETRiS project. Thanks to Josefine Asmus and Ivo Sbalzarini for collaboration on the work on design centering, which was very insightful. I also thank Sergio Siccha, for taking our friendly discussions so seriously that we ended up collaborating in the mapping symmetries work. Thanks also to Robert Wittig for a beachside discussion at Samos that led to a collaboration on the model-based approaches to 5G. I thank Arka Maity, Nishant Budhev and Tulika Mitra for sharing their LTE traces with us.

I started this Ph.D. at the cfaed cluster of excellence, which provided funding and a great academic environment. I want to thank everyone at the program office for helping me throughout this time, as well as my thesis advisory committee, Jeronimo Castrillon, Christel Baier and Hermann Härtig. I also want to thank Conny Okuma for her patience throughout the years with my incomplete formularies and late handing over of documents. Thanks as well to the German Research Foundation DFG for funding me after cfaed.

The final phase of my Ph.D. was mainly funded by the Studienstiftung des deutschen Volkes. Besides financial support also provided me with an excellent offer of intellectual complementary opportunities. Thank you for this opportunity, and thanks to Maike Lieser for helping me apply to this scholarship, I am sure I would not have received it without her help. I also want to thank her for everything else, as she was probably the biggest positive influence in my life during the time of my Ph.D.

I want to thank everyone at TEDxDresden and everyone from animal rights activism for giving me meaningful projects to do with my life besides my research. Also to everyone at Bodyworks and Basketball Club Dresden for giving me a constant outlet to find a healthy balance with sports.

Finally, and most importantly, I want to thank my friends and family for being there for me and reminding constantly of all the important and enjoyable aspects of life, besides academics. To all my friends in and around Dresden, who accompanied me through life these past six years, thank you for making this one of the best times of my life. To my friends back in Aachen, San Salvador and spread throughout the rest of the world, thanks for being a constant source of love and friendship that has kept me grounded. I won't list everyone who has made my life better these last six years and whom I consider a friend, I'm sure they know who they are, and I thank each and every one.

I would certainly not be who I am, and this thesis would not be possible, without the tremendous support from my family. My cousins, uncles and (great) aunts, my two big sisters, thank you everyone for always being there for me. Especially my little sister Ute, who's accompanied me a large part of my time here in Dresden, being a constant source of support and inspiration. My father made me be curious and think critically since I was a kid, and coupled this inspiration with unconditional love, which I am certain was an indispensable for me to write this thesis. My mother made me be social and empathic, and made sure I became a well-rounded person. Her constant support and openness made me always do what interested me, and I am certain this thesis would never have happened without her. Thank all of you for everything!

Andrés Goens, January 2021

CONTENTS

1

INTRODUCTION

Programming computers is notoriously difficult. Indeed, people learning to program usually struggle, paradoxically, with the fact that the computer does precisely what they tell it to do. This is confusing, not because a computer program is executed faithfully, but rather, because humans think at a very different level of abstraction.

It is certainly true that instructions in computer architectures are at a completely different level of abstraction than the instructions we give each other. However, most programs are also not written at the level of the architecture. Programming languages are designed with increasingly improving abstractions, making it easier for programmers to express themselves. Complementary to these efforts are compilers, which serve as bridge between the levels of abstraction. Ideally, a compiler translates the abstract human-level expressions into efficient machine-level instructions. While we have made significant progress, this task has proven to be dauntingly difficult.

Traditionally, we have put the research and effort into optimizing the execution of a single core. Most of the progress of decades of research in programming language and compilers revolves around this single-core model. In the last decade or two, however, with the multicore era, this challenge has increased dramatically. Now we have to use and coordinate multiple cores, commonly with different capabilities. The widespread programming language abstractions and compiler analyses of today are ill-suited to tackle this challenge.

There is probably no universal solution to these emerging problems, as different domains have different requirements. This thesis thus focuses mostly a particular domain, that of Cyber-Physical Systems (CPSs) or generally, embedded systems. In this domain, a family of methods called software synthesis seeks to enable efficient programming of complex multicore systems. Central to these methods is a focus on using models for describing computation. We follow the idea of letting theory inform practice, in striving to improve methods of software synthesis. We do this by identifying and exploiting mathematical structures in the problems in software synthesis.

1.1 The Multicore Era

On the hardware side, the last two decades have firmly established what we call the multicore era. Modern computing systems are almost universally composed of multiple logical cores, and there is a clear trend of increasing both the number and the degree of heterogeneity of these cores. This increasing complexity brings about an increasing challenge in taming it.

Both the execution frequency and the closely intertwined single-core processing speed of computing systems increased exponentially up until the early 2000s (cf. Figure 1.1), an empirical fact observed by Gordon Moore in 1965 [Moo+65]. Since the early 2000s, however, while transistor sizes continue to decrease, the exponential frequency scaling has

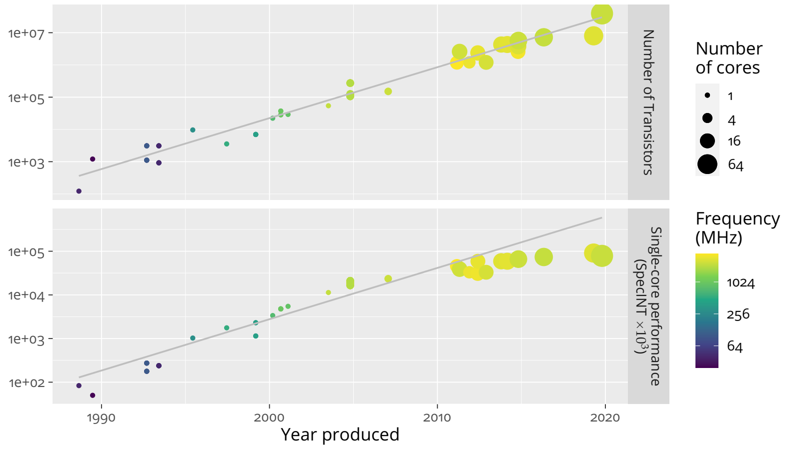


Figure 1.1: Chip trends as obtained from [Rup20]. The lines present the exponential growth prediction if considering data up until the year 2000.

stopped (cf. Figure 1.1). Individual and clever designs have continued to improve single-core performance, albeit at a significantly slower rate. Instead, most improvements in microchips in raw processing power have come from a paradigm switch. Hardware designers resorted to develop multicores, microchips composed of multiple cores that can execute in parallel. Additionally, since different use-cases benefit from different computing core architectures, the inclusion of multiple chips has paved the way for heterogeneity. This has resulted in a multicore era of computing.

A programmer writing a piece of C code in the early 1970s would automatically benefit from the increasing processing speeds. By recompiling her code for a processor $10\times$ faster, she could very roughly expect her code also to run about $10\times$ as fast. This has changed. A programmer writing a piece of (sequential) C code in the early 2000s cannot expect anywhere near a 100-fold increase in performance today when she uses a microchip with $100\times$ the number of transistors. At least not without fundamentally restructuring her code to exploit parallelism. Contemporary systems aimed at performance almost ubiquitously consist of multicore chips, in many cases heterogeneous, which tremendously increases the system complexity. It is a clear trend that the complexity and heterogeneity will continue to increase [Völ+16; Cas+18].

As the number of cores in a chip keeps increasing, the importance of coordination and communication between the cores raises as well. Memory latency and bandwidth has been a bottleneck for many classes of applications for a while. In the case of manycores, which are multicores where the number of cores goes over several dozens, even to hundreds or thousands, the on-chip memory subsystem becomes a central design point of the chip. Systems based on Network on Chip (NoC) technology become necessary, lest a single bus interconnect becomes the bottleneck of the system when thousands of cores want to communicate simultaneously through it.

In fact, for a multitude of reasons, manycores are commonly designed in a hierarchical fashion. Smaller clusters of cores locally interconnected communicate between each other and off-chip via a larger NoC. These clusters and systems are usually heterogeneous as well, like the Karlay MPPA3 Coolidge [inc20], with includes accelerators for cryptography and

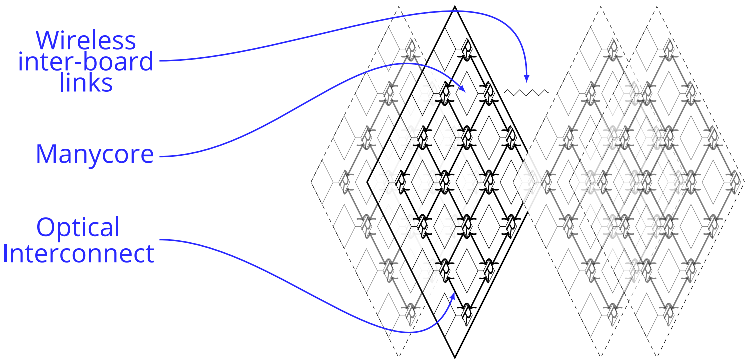


Figure 1.2: The HAEC architecture [Fet+19] has multiple levels of hierarchy: on-chip, intra-board (optical links) and inter-board (wireless).

secure cores, alongside general purpose cores. Some systems even propose multiple layers of hierarchy, like the Highly-Adaptive Energy-Efficient Computing (HAEC) topology [Fet+19], depicted in Figure 1.2. This is a proposed 3D-stacked system with multiple printed circuit boards (PCBs) each with multiple manycore chips, not a single chip with this complex interconnect. However, this design could allow for very low latencies, such that challenges of programming it are comparable to those of programming such a topology as part of an on-chip memory subsystem. What happens if the mulicores in this system are similar to the Karlay MPPA3 Coolidge? This would yield more than 5000 heterogeneous cores connected at five levels of hierarchy in different topologies. More generally, complex topologies, with possibly multi-level hierarchies, add a layer of complexity to modern manycore systems besides heterogeneity and concurrency. The techniques used for programming a system with two or four identical cores are mostly not suited at all to program these dauntingly complex topologies. We need to re-think our design process to accomodate for these changes in complexity.

1.2 Programming Multicores

As already mentioned, programming is notoriously difficult since it translates from the level of abstraction of human interactions to the instructions of a computing system. The multicore era greatly aggravates this already difficult problem. Reasoning about concurrency, parallelism and heterogeneity on top of the correct functionality of software is significantly more difficult for a programmer. In general, the productivity of developers cannot keep up with the pace of developments in hardware. We speak of a “software productivity gap” [EMD09; CL14].

When programming multicore systems, abstractions that proved very useful for programming single-cores become inadequate. This is exacerbated by a problem with the legacy of ISAs. They are usually based on machine models which are also sequential at the instruction level [Chi18]. A universal concept in programming is that of repeating an action multiple times, or looping. For example, consider the task of iterating through a bunch of pictures and determining which of them contain cats. For instructing a human, we can probably say something like “look through those pictures and sort out the ones that have cats”. A modern x86 chip, on the other hand, would understand something closer to this:

```
LBB0_1:
  cmp         dword ptr [rbp - 56], 10
  jge         LBB0_4
  mov         edi, dword ptr [rbp - 56]
  call         _read_picture
  mov         qword ptr [rbp - 64], rax
  mov         rdi, qword ptr [rbp - 64]
  call         _contains_cats
  movsxd         rcx, dword ptr [rbp - 56]
  mov         dword ptr [rbp + 4*rcx - 48], eax
  mov         eax, dword ptr [rbp - 56]
  add         eax, 1
  mov         dword ptr [rbp - 56], eax
  jmp         LBB0_1
LBB0_4:
  mov         eax, dword ptr [rbp - 52]
  mov         rcx, qword ptr [rip + ___stack_chk_guard@GOTPCREL]
  mov         rcx, qword ptr [rcx]
  mov         rdx, qword ptr [rbp - 8]
  cmp         rcx, rdx
  mov         dword ptr [rbp - 68], eax
  jne         LBB0_6
```

This snippet is a very oversimplified version of the task, but it serves to make the point. Where we abstractly tell a human to look through the pictures, and they understand them as a whole set, interpreting themselves how to go through the set. On the other hand, we instruct the machine to iterate through them by a series of very fine-grained commands. We need to set certain registers to contain the right memory addresses, before calling an instruction to operate on them. We then call external functions that do the reading and cat identification. To then loop through the pictures here, simplified, we repeat this reading and identifying by jumping to a previous point in the sequence of instructions. Even this x86 assembly snippet is already an abstraction, not only because it uses human-readable mnemonics for the instructions, but more so because it also abstracts away the concrete memory addresses and the microarchitecture. In practice, however, almost no one would write this assembly code. Instead, they could write something closer to this (equivalent) C snippet:

```
for(i = 0; i < N; i++){
    char *f;
    f = read_picture(i);
    results[i] = contains_cats(f);
}
```

Notice how the register management and several other low-level details are abstracted away. The end of the loop is very clear to read, as we know when we have reached the final picture. We can certainly say this is at a level of abstraction between the human and machine instructions listed above. However, the very widespread `for` instruction we used here also has the inherently sequential semantics, as exhibited by the assembly code it translates to. The semantics of the for loop are that the loop body will execute completely. After each iteration of the body, the increment expression is executed (usually incrementing the iteration variable), and the condition is evaluated, deciding wether to continue iterating. Indeed, in the two (equivalent) snippets above, we do not know how the functions `read_picture` and `contains_cats` work. Do they have an inner state, or side effects? We do not know if we can call `read_picture` in a different order, or multiple times in parallel. Perhaps it is internally keeping a single reference to the iterator of the image files and doing so would break the logic. The `for` instruction is very useful to abstract away the logic of registers and instruction jumps, but not a useful abstraction for expressing concurrency. A similar construct exists in functional programming, `map`, which generally does not have this implicit sequential semantics. The `map` instruction is what is called a higher-order function, taking a function as an argument and applying it to a list or any iterable object, in general. The same cat-identifying snippet, in Haskell, can be written as follows:

```
result = map (contains_cats . read_picture) pictures
```

While the language separates statefull and stateless computation, allowing a great analysis of concurrency, there are reasons why Haskell is not the most widespread language for embedded systems. For example, garbage collection makes execution times very unpredictable. Similarly, the lazyness of the language adds a performance penalty to large complex computations. Compiling Haskell code to an efficient single-core execution is significantly more challenging than equivalent C code. The lazyness also makes it difficult to reason about time in the computation. This is crucial in application domains like CPS, where the systems interact with their environment. The `map` abstraction, as implemented in Haskell, is not well-suited for many tasks in the domain of CPS. In general, we are faced with trade-offs between abstract expressivity and translatability to an efficient execution. At its core, the challenge is about choosing the right models and corresponding abstractions for a particular domain.

1.3 Software Synthesis

Models play different roles in science and engineering. E.A. Lee explains this well in [Lee17]. He argues that scientists adapt their models to fit experiments in the world, while engineers adapt designs in the world to fit their models. Indeed, some fundamental principles of computation, like λ-calculus, are arguably discovered instead of invented, as Wadler contends [Wad15]. Those might fit in the first paradigm, giving computer science a justification for its name. In the case of programming multicore systems, however, the problem is clearly in the second realm: we need to engineer good models [Lee06]. No serious argument can be made for languages like C or Haskell, nor the x86 instruction-set; They were invented, not discovered.

There are different ways of finding and exploiting the right models for programming multicores. It is unlikely that there is a single right model for this. Different models are differently suitable for different use-cases. For example, applicative functors in functional programming [Mar+14] seem to be a great model for expressing I/O concurrency in microservice-based systems. As mentioned before, however, Haskell and its underlying model are not a great fit for CPS.

For CPS and, embedded systems in general, there is a family of methods called software synthesis [RPM92; Abb+93; Lin98; BLM00; Pin+95; CSL11; BML12]. It is a family of methods devised precisely to help with the burden of fully exploiting the capabilities of modern multicores. Inspired by hardware design flows, it aims to bridge the ensuing (software) productivity gap by integrating knowledge of the application and target multicore architecture into the compilation process. At the core of these methods lies a shift in the programming model. Instead of the de facto sequential, shared-memory model, programmers express the code in diverse Models of Computation (MoCs). This makes the underlying model explicit, not implicit as is the case in most programming languages.

These models expose the structure of the computation in ways that permit a compiler to reason about its parallel execution, even in the presence of heterogeneous hardware. Aided by abstract models of the target architecture, we can design compilers for multicore systems that devise execution strategies specialized to the target architecture and applications. Depending on the flow, the target architecture can be implicit in the methodology [RPM92] or be an explicit input to the flow [CLA11]. This can be realized for example by finding efficient mappings, i.e. allocations of computational and communication resources to the different parts of an application.

As mentioned above, the central principle behind software synthesis is the underlying model of computation. Some approaches [Lin98] use general models, like Petri Nets [Pet62], while others [RPM92] more constrained models like Synchronous Data Flows (SDFs) [LM87]. Most allow for multiple models [BLM00; Pin+95; BML12], generally dataflow models. Making the model explicit just makes it easier to see the trade-off between expressivity and translatability to an efficient execution. The advantage of models like Petri nets is that they can express virtually any computation. On the other hand, very constrained models, like SDF provide behavioral guarantees that permit several optimizations, like static schedules and channel bounds [Par95].

Several more modern flows [Thi+07; CLA11; PEP06; Kan+06] have settled at the Kahn Process Network (KPN) model. Originally meant as denotational semantics for parallelism [Kah74], the model has been shown to be compatible with dataflow [LP95]. Kahn Process Networks are provably deterministic [Kah74], which is not the case for other models, e.g. Petri Nets. In a canonical sense, KPNs are more general than most dataflow models, and represent the most general deterministic dataflow model of computation [LM09]. In this thesis we will focus on a software synthesis flow [CLA11; CL14] based on KPN.

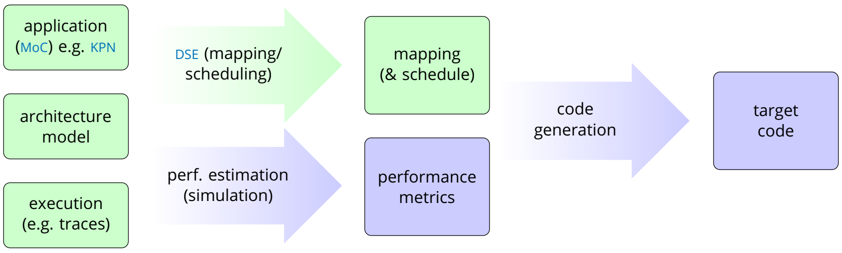


Figure 1.3: A flow for MoC-based Software Synthesis. The main abstractions colored in green are the ones we deal with in this thesis.

Figure 1.3 shows the general structure of many such MoC-based flows. Application, architecture and execution models are used for performance estimation and Design-Space Exploration (DSE), commonly intertwining these two processes, to produce performance metrics and a mapping (and schedule) for the application on the target architecture. A final, code generation step generates code for the target architecture. This code is usually software for an existing architecture, but can also be hardware (e.g. in [Hau+08]), in flows that are closer to High-Level Synthesis (HLS), which is the inspiration for the name software synthesis in the first place. This thesis is limited in its scope, and it does not deal with the whole flow as depicted in Figure 1.3. It deals mostly with the abstractions on the left side of the figure, colored in green. Chapter 2 will review these abstractions, including some of the models of computation above, and their relationships. MoCs are reviewed more generally in Chapter 6.

1.3.1 *Problems*

The methods of software synthesis are a promising avenue for programming multicores, but they still have multiple problems that make them impractical.

Consider a concrete KPN-based instance of the flow depicted in Figure 1.3, like [CLA11; PEP06; Thi+07]. KPN is a well-defined MoC, used for specifying and reasoning about applications in the flow. However, the architecture or the execution, which are comparably important, are commonly modeled in an ad hoc fashion. With a specific architecture or topology in mind, the flows encode the architecture model with a series of performance metrics or similar numbers characterizing a specific hardware, while the structural properties are implicit in the problem formulations and algorithms.

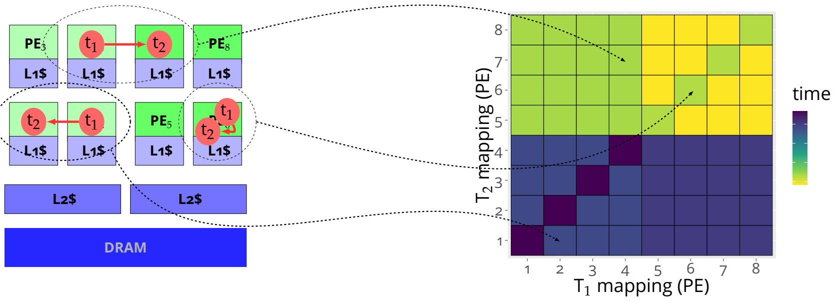


Figure 1.4: An example of the mapping space for a simple two-task application.

Consider the very simple example of mapping two processes t_1, t_2 where t_2 depends on t_1 onto an off-the-shelf multicore like Odroid XU4, which features a Samsung™Exynos 5 chip with an octacore (4+4) ARM big.LITTLE™, as depicted in Figure 1.4. The mappings are plotted by encoding the mapping of each of the two tasks as the x and y coordinates of the grid, and the color of the squares in the grid encodes the execution time of the application with that mapping. The example is very simple and we deliberately chose it such that it can be easily visualized. In particular, the actual values of the execution time are irrelevant for this discussion and have been omitted. Still, many of the problems are clear with this simple example already.

There are exactly $2^8 = 64$ mappings in the mapping space, yet only 6 distinct execution times (colors). This is because, at least a priori, mapping both tasks to PE_6, as shown in the figure, or mapping them both to PE_7, will obviously result in the same execution time, since the two cores are identical (Cortex-A15™). This can be generally understood as a property of the *symmetries* of the architecture, and should be exploited when exploring this mapping space.

Similarly, researchers often use heuristics based on geometric properties of the design space to explore it. Yet they often disregard the encoding they use for the design space. If we consider the point $(4, 4)$ on Figure 2.9, there are four points adjacent to it, yet they are vastly different in terms of their execution time. We can compare the geometry of the space with the geometry of the architecture itself, and see why this is the case: the distances in this space do not reflect the architecture with its heterogeneity and its memory subsystem. In general, the geometry of this space does not reflect the geometry of the problem.

Mappings encode the resource allocation for the application to an architecture. As such, they inherit structural properties of both the application and its semantics, as well as of the underlying architecture. Yet mappings are commonly treated as simple lists of assignment, disregarding this structure, like which tasks depend on which, or if they are mapped to cores with a large communication latency between them. Mapping algorithms commonly encode a heterogeneous architecture as a list of numbers of cores of different types, or perhaps use a grid system to encode processing elements (PEs) as they assume a NoC with a regular-mesh topology. These models break down as soon as the complexities of the architecture transcend the fixed model, for example by having multiple clusters or levels of hierarchy, or star-mesh topologies instead of regular meshes.

The problems mentioned here permeate the design of the internal algorithms in software synthesis flows, which effectively constraints them to a small class of models or disregards opportunities for reasoning about the structure of the problem. While memory has been identified as a first-class citizen for achieving efficient implementations, many methods also consider it just as an afterthought. For example, when discussing heterogeneity in architectures, the heterogeneity implied by the memory subsystem is seldom considered, nor are emerging memory technologies like non-volatile memories (NVMs).

The issues raised above are not inherent issues with the flow, but rather with the state of practice. However, the flow itself does has some inherent limitations as well. The KPN model of computation falls short on certain use-cases. For example, the blocking-read semantics common in KPN implementations are ill-suited for certain cases of data-level parallelism. Also, perhaps more importantly, KPNs do not model time in the physical world, which plays a central role for the execution of CPSs. In general, a model-based design approach needs to evolve its models according to the use-cases.

Another inherent problem with the flow as formulated is the structure of the flow itself. An application is described using a concrete MoC and then this is used to reason about an implementation. However, the flow as depicted in Figure 1.3 (and implemented in practice in many instances) disregards transformations at the level of the application. This could mean a feedback loop back to the application, or perhaps semantics-preserving code transformations at the model level, as part of the exploration.

If methods like software synthesis are to be used in practice, we should also make sure they also work in practice. Strong results on a varied benchmark suite from real-world applications are usually a much better indicator for practical applications than, say, a good asymptotic worst-case behavior. In order to get such results, however, we need such a varied realistic, up-to-date benchmark suite. In reality, however, increasingly branching subdomains and concerns of intellectual property mostly yield a scarce landscape of outdated benchmarks instead.

Finally, there are multiple issues with these flows that depend more on the industry itself than the methods directly. Tool support and maturity, degree of adoption and knowledge of the models are all beyond the realm of the academic contribution of this thesis.

1.4 Contribution

In this thesis we seek to improve the tools we use for understanding and tackling the problems discussed with software synthesis. We work in a model-based perspective and consider the trade-off we have introduced, between abstract expressivity and translatability to an efficient execution. To consider this we tackle the problem from both sides: the models and the compilers, in a very general sense, that translate to an efficient execution. The main idea behind this thesis is that the underlying models endow the problem with structure. We can then identify this structure (mathematically) and leverage it to improve our solutions. Again there are two ways of doing this:

1. by taking a concrete flow and improving it leveraging its own structure, or
2. by changing the underlying models in a way that improves the balance in some way in the trade-off above.

This thesis discusses both. We first focus on software synthesis for (high-performance) embedded systems running on Multi-Processor Systems-on-Chip (MPSoCs). In particular, we focus on a concrete software synthesis flow [CLA11; CL14] based on KPNs. Chapter 2 introduces this flow, as depicted in Figure 1.3, and the corresponding background on the mapping problem.

To evaluate methods in software synthesis in particular and compilers in general, we need to test them on benchmarks. Chapter 3 discusses benchmarking in compilers, and introduces some benchmarks we use in the thesis. It also discusses benchmark generation, with its advantages and pitfalls, both using random processes and machine learning.

As motivated in Figure 1.4, the mapping problem in software synthesis has a rich structure, like its symmetries or geometry. We identify and describe this structure in Chapter 4. Describing the structure is only as useful as the applications we find from it. In Chapter 5 we discuss multiple applications, e.g. at compile time in DSE or at run-time in hybrid mappings. We also show how this structure can be used to formulate other properties of mappings, like robustness or compactness, which can be useful for resilient computation even in real-time scenarios in CPS.

After exploring how to improve concrete flows with its structure, we turn our attention to the underlying models. Chapter 6 reviews Models of

Computation (MoCs) in general, and shows how to improve the methods here. We first show how to improve existing methods, discussing what we call the MacQueen gap in the KPN semantics. We discuss a novel model, Reactors, where time is a first-class citizen, and discuss applications in the telecommunications and automotive domains.

MoCs are abstract mathematical models, they need to be exposed to programmers using a language or an API. In Chapter 7 we discuss the programming languages used to develop MoC-based applications. We review different existing languages, including the Ohua paradigm, which can be used for implicitly defining dataflow applications. We discuss language-level transformations and abstractions in the context of Ohua and how MoC-based design can be used for optimizing I/O in microservice-based architectures, i.e. in a collection of loosely-coupled services in a networked setting.

The topic of this thesis is broad, and much related work exists for all aspects covered here. While different chapters cover related work pertinent to the topic discussed, we review and discuss it concisely again in Chapter 8. Finally, some conclusions from this work are summarized in Chapter 9.

While all topics covered in this thesis are related by model-based design of software, not every chapter depends on everything previous. Figure 1.5 shows the logical dependencies of the chapters, and in some cases, the sections of the chapters in this thesis. Any path in this graph should yield a consistent exposition of the topics discussed. A reader only interested in some topics can readily skip chapters and sections that are not in the path to the sections that interest them.

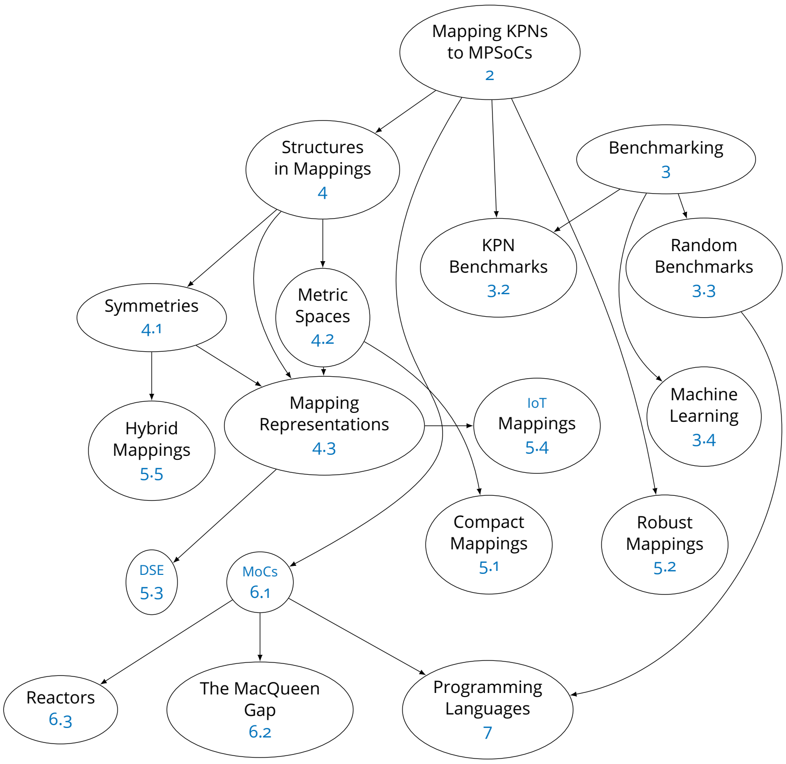


Figure 1.5: Dependencies of chapters and sections of this thesis.

1.4.1 *A Note on Originality*

This thesis presents the fruits of over half a decade of research on the subjects presented. Research, especially in an interdisciplinary approach like presented here, is much more fruitful when collaborative. In the case of joint work, I have made an effort to focus only on my own contributions in this thesis, whenever possible. I have also taken care to describe the work of my colleagues as theirs, when I have included it as an indispensable requirement to understand my own work. However, some of the ideas in this thesis are the result of joint work and cannot be credited to a single person. In those cases I have also taken care to describe the work as joint and mention other coauthors. If in doubt, any idea or result that I have included here which has already been published elsewhere is also due to my coauthors.

2

MAPPING KPNS TO HETEROGENOUS MPSOCS

Software synthesis refers to a family of methods, rather than a concrete one, which share common properties about the abstract flow for generating code for an efficient execution in (heterogeneous) multicores. It can be seen as embedded in a spectrum of design approaches going from hardware design (and classical Electronic Design Automation (EDA)) through hardware-software co-design up to software synthesis on the other end. While some principles apply more generally than others, to actually produce and optimize code, we need to focus on a concrete flow. In this chapter we will introduce the concepts behind software synthesis and mappings in a concrete flow, mapping KPN applications onto heterogeneous hardware. The flow is an instance of the general flow from Figure 1.3, and is presented in detail in [CL14].

As is general in Software Synthesis, the applications to be executed are represented abstractly, linked with a model of computation, Kahn Process Networks (KPNs). Similarly, the target architecture is assumed to be known at compile-time, and is modeled via an abstract architecture model. The KPN model has a property that allows to capture the abstract execution behavior in a trace that is independent of the execution target. Combining these application and architecture models, and using an execution trace, a simulation can be used to estimate the performance of a mapping - an assignment of physical execution and communication resources on the target architecture to the logical (abstract) components of the KPN application. In an iterative process, these estimations can be leveraged to determine a near-optimal mapping subject to objective goals (e.g. execution time, energy consumption). Finally, a compiler can lower the KPN application to an executable that uses the selected mapping.

The rest of this chapter will explore the various models referred to in this flow, with precise mathematical definitions and a discussion of common design choices and goals.

2.1 Kahn Process Networks

The main flow we investigate in this thesis is based on the MoC of Kahn Process Networks (KPNs). In this section we introduce this model, or rather, its most common implementation with blocking-read semantics [KM76] . In Chapter 6 we will discuss the original (denotational) semantics [Kah74] and how they differ to those introduced here. There, we also discuss other MoCs and how they relate to each other.

We can think of a KPN as computation distributed among different *processes* (originally derived from coroutines). Each of these processes executes sequentially and is Turing complete. However, the processes share no memory, they have local memories accessible only to themselves. They communicate between each other using *channels*. These channels work as unbounded FIFO buffers. Processes have sets of outgoing and incoming channels. As an instruction, any process can write to one of its outgoing channels or read from one of its incoming channels. They do so in discrete tokens of data.

```
1  __PNkpn fft_process
2    __PNin(int cnt, short src_data[N])
3    __PNout(complex freq[N]){
4       int i, loop_cnt;
5       __PNin(cnt)
6          loop_cnt = cnt;
7       for(i = 0; i < cnt; ++i)
8         __PNin(src_data) __PNout(freq)
9          fft(src_data, freq);
10 }
```

Listing 1: An Fast Fourier Transform (FFT) implemented as a KPN process in CPN, based on Appendix A.1.3 of [CL14]

The original language [KM76] was proposed as an extension of POP-2, which is pretty dated and has fallen out of use today. Instead of this language, we will consider a more modern incarnation, C for Process Networks (CPN), which extends the C programming language [She+14]. We do so by looking at the example from Listing 1. Processes in CPN are instantiated from process templates, similar to classes and objects in object-oriented languages. The listing shows a very simplified process template for an FFT process. Lines 2 and 3 declare the incoming and outgoing channels for the process. In Line 5, the `cnt` channel is read and its value is stored in the local variable `loop_cnt` in Line 6. Then in Lines 7-10 the process applies an FFT to the data in its incoming channel `src_data` and outputs it to an outgoing channel, `freq`. Similar to the read operation in Lines 5-6, the values of the input channel data are available in the identifier `src_data` in the scope of the `__PNin`. In an analogous fashion, the values written to the `freq` variable in the scope of the `__PNout` are written to the corresponding output channel.

In general, the communication in KPNs is asynchronous: When a process writes to an outgoing channel, the data is buffered in the channel until it is read, and the process continues to execute. If a process reads from a channel, it receives the oldest token buffered in the channel. If there are no tokens, execution blocks until such a token is written to a channel - hence the denomination of blocking-read semantics. A channel can be the outgoing channel of at most one process (this should also be so for at least one process, otherwise the channel is useless). On the other hand, if a channel is an incoming channel to multiple processes, all tokens are copied for each of those processes. Hence, all processes will see the exact same incoming stream of tokens from a shared channel, instead of splitting them up.

Let us consider the FFT process from Listing 1 and combine it with other processes into a full application. Listing 2 describes a simplified algorithm for a low-pass filter on a stereo sound file, using this FFT process. We also omit the templates and channel declarations in this simplified listing. The `src` process reads the stereo file, splits it into two channels and sends the sound in blocks of a determined length as tokens. These files are then transformed from the time domain to the frequency domain using an FFT, filtered and transformed back to the time domain. A sink channel gathers the filtered blocks from both channels, left and right, and combines them again into a stereo sound file that can store.

```
__PNprocess src = src_process
  __PNout(cnt, src_l_out, src_r_out);
__PNprocess fft_l = fft_process
  __PNin(count, src_l_out) __PNout(fft_l_out);
__PNprocess fft_r = fft_process
  __PNin(count, src_r_out) __PNout(fft_r_out);
__PNprocess filter_l = filter_process
  __PNin(count, fft_l_out) __PNout(filter_l_out);
__PNprocess filter_r = filter_process
  __PNin(count, fft_r_out) __PNout(filter_r_out);
__PNprocess ifft_l = ifft_process
  __PNin(count, filter_l_out) __PNout(ifft_l_out);
__PNprocess ifft_r = ifft_process
  __PNin(count, filter_r_out) __PNout(ifft_r_out);
__PNprocess sink = sink_process
  __PNin(count, ifft_l_out, ifft_r_out);
```

Listing 2: An audio filter KPN application in CPN, based on Figure 7a in [She+14]

The data flow in the example of Listing 2 is very structured: it goes from the source, splits into two channels, through the filter, back to the sink. This structure can easily be visualized in a graph, like in Figure 2.1. More generally, we can think of any KPN application as a directed graph $K = (V_K, E_K)$, where the nodes V_K represent the processes, and the edges E_K, the channels. This works even when a channel is an incoming channel for multiple processes. In that case, we can split it into multiple edges from the process it is going from, to each of the target channels. We can do so without loss of generality since these are the semantics of such channels. We call this graph the *KPN graph*.

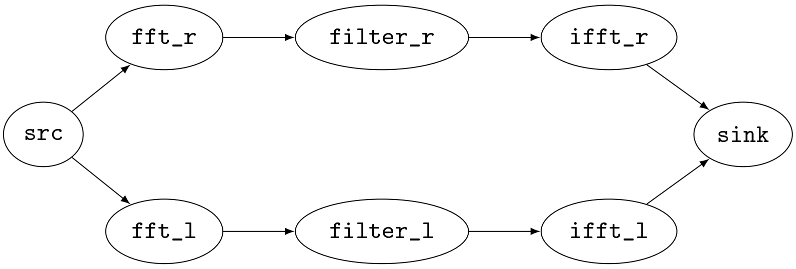


Figure 2.1: The audio filter application as a KPN graph

2.2 Execution Traces

Kahn Process Networks have a more abstract definition with mathematical semantics [Kah74], in the sense of Scott [Sco70]. These abstract away the concrete implementation of individual steps in a computation. Even so, the execution of a computation can be thought of as a series of steps or partial computations that eventually yield the final result. These series, which is commonly referred to as execution trace, can be captured as a sequence of steps, e.g. as the element of a Scott Domain[1]. Abstract computations, modeled as Scott-continuous functions, can can make computa-

---

[1] This will be discussed more in-depth in Chapter 6

tions of arbitrary length. For an alphabet Σ, this is modeled by (countably) infinite sequences in $\Sigma^\omega := \{(a_n)_{n\in\mathbb{N}} \mid a_n \in \Sigma \text{ for all } n \in \mathbb{N}\}$. A concrete execution, on the other hand, always has a finite length. It always resides in Σ^*, the Kleene closure of Σ. For a (Scott-continuous) function, this sequence can be modeled as a finite string in the computation domain.

In a concurrent execution, multiple entities concurrently execute steps. As modeled by Kahn, these entities all implement individual functions. As such, there is not a unique series of steps that can be said to be the execution trace of the computation. To see this, consider the example depicted in Figure 2.2: It shows multiple execution orders for the audio filter KPN application. If we were to consider the values in the channels, each of these orders would yield a different sequence of values. In this case, the actions in the alphabet Σ should also model the actual values of the arrays of floating-point values that can be stored in the channels, which is why we show the processes in the figure instead. The traces corresponding to the executions shown in Figure 2.2 are all equivalent.

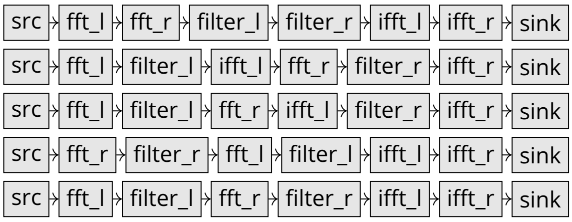


Figure 2.2: Different possible sequential executions of the audio filter KPN.

In the case of a concurrent execution thus, traces are in fact equivalence classes of strings. We define this more formally, following [Maz95], the first chapter of [DR95]. Let Δ be a symmetric, reflexive relation on Σ, which we call a dependency. This means that if $(a,b) \in \Delta$, we have $(b,a) \in \Delta$ and also $(a,a) \in \Delta$ for all $a \in \Sigma$. With Δ we define an additional relation over Σ, namely $I := (\Sigma \times \Sigma) \backslash \Delta$. We call I the induced independency. We define an equivalence relation $\sim_I$ on the monoid Σ^* (with respect to concatenation) as follows: We require that if $a, b \in I$, then $ab \sim_I ba$. The relation $\sim_I$ is defined as the least congruence that satisfies this requirement. Note that a congruence is an equivalence relation that respects the algebraic structure, in this case the monoid structure of the concatenation operation. We call the equivalence classes of $\Sigma^*/\sim_I$ traces. By definition, the concatenation operation on Σ^* factors over the equivalence relation $\sim_I$, and thus $\Sigma^*/\sim_I$ defines a monoid (with identity element $[\epsilon]_{\sim_I}$, where $\epsilon \in \Sigma^*$ is the empty string). We call this the Trace Monoid, $T(\Sigma)$. We care about the algebraic structure of a monoid since it is central to the definition of Scott-continuity.

There are two additional equivalent definitions of this monoid as histories and dependence graphs. We present histories here, as they are better for the intuition. Instead of a single alphabet Σ, we have a finite set of alphabets $\Sigma := (\Sigma_i), i \in \mathcal{I}$, where $\mathcal{I}$ is a finite index set. We can think of the indices as corresponding to the entities in the system (e.g. processes) and the alphabets Σ_i to the alphabets of actions of these individual entities. If we think of the individual entities as computing some function, their execution trace will be a unique string $a_i \in \Sigma_i^*$ (recall that concrete executions are finite). Since, in general, these entities do not compute independently, they have common synchronization points. These synchronization points are abstractly modeled in the computation alphabet by mutual elements in $\Sigma_i \cap \Sigma_j$ for two entities $i, j \in \mathcal{I}$. In the case of synchronous dataflow[LM87] application, for example, we could model the alphabet as being tuples of a channel and a value, and the common synchronization points would be reading to or writing from a value. In KPN, since the communication is asynchronous, we would need to model both the channels and the processes as entities.

We can define a monoid, the product monoid $P(\Sigma)$, by component-wise concatenation of the strings: $(a_i)_i(b_i)_i = (a_i b_i)_i$ for all $i \in \mathcal{I}$. However, not every such a string product can be the history of a system. The synchronization points of different subsystems should be consistent with each other. To avoid this, we want to ensure histories are consistent. For this, we define elementary histories as follows: For any $a \in \bigcup_{i\in\mathcal{I}} \Sigma_i$, the elementary history of a is the tuple $(a_i)_{i\in\mathcal{I}}$, with

$$a_i = \begin{cases} a, & \text{if } a \in \Sigma_i, \\ \epsilon, & \text{otherwise.} \end{cases}$$

Here, ϵ represents the empty string. The monoid generated by all elementary histories for elements in $\bigcup_{i\in\mathcal{I}} \Sigma_i$ is called the history monoid $H(\Sigma)$, and is a submonoid of $P(\Sigma)$. If we examine the definition, it is not difficult to convince ourselves that these are precisely the histories which avoid inconsistencies.

We can go from a trace to a history by the morphism $\pi : T(\bigcup_{i\in\mathcal{I}}) \to H(\Sigma), a \mapsto (\pi_i(a))_i, i \in \mathcal{I}$, where π_i is the projection $\bigcup_{i\in\mathcal{I}} \Sigma_i \to \Sigma_i$. Here, for the trace monoid $T(\Sigma)$ we define the dependencies to be $\bigcup_{i\in\mathcal{I}} \Sigma_i \times \Sigma_i$. This is not just a morphism, but in fact an isomorphism: See Theorem 1.5.4 of [Maz95]. Thus, the two concepts are equivalent. For the rest of this thesis we will use the terms traces and histories interchangeably.

Traces, and equivalently histories, can be used to describe the concrete computations in concurrent systems like those described by a KPN. They are also well-suited to model these systems in the context of process calculi, like Communicating Sequential Process (CSP). However, an important observation is the converse: a concrete execution of a KPN is determined uniquely by its history. Moreover, any concrete implementation of the KPN realizing the same execution will have the same history: the history is an invariant of the abstract execution model. It captures the concurrent essence of the concrete computation.

2.3 Architecture Models

Hardware architectures are in contrast to applications from the point of view of modeling. Abstraction boundaries are arguably more clearly defined in the hardware world: semiconductor components like transistors implementing digital switches are used to form logic gates (like a NAND gate). Logic gates are used in increasingly complex logic diagrams for building components like an Arithmetic Logic Unit (ALU). These components are combined into digital machines in a microarchitecture to expose a well-defined Instruction-Set Architecture (ISA) in a PE [Lee17]. PEs can then be connected via on-chip interconnects to on-chip memory and other peripherials to make an MPSoC. There are mostly clear boundaries between these *platforms*, as A. Sangiovanni-Vincentelli calls them [San07] (which are levels of abstraction). Designers at each level expose a small amount of complexity through these established abstractions, in what is commonly referred to an hourglass design [Bec19].

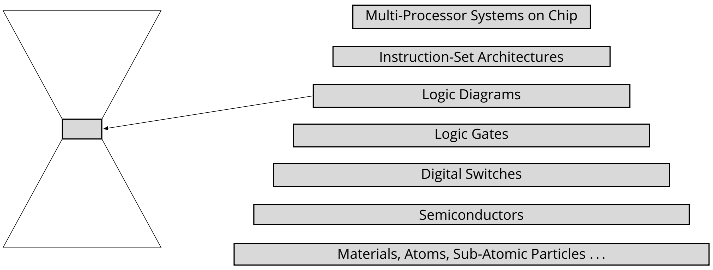


Figure 2.3: Different levels of abstraction in architectures

Figure 2.3 summarizes different models used at different levels in architectures. The “bottleneck” design shown on the left implies how there are well-defined abstraction layers at the different levels. The layer between hardware and software is, in a sense, also just such a layer of abstraction. Since these layers are clearer in the hardware world, so are the corresponding models at those levels of abstraction. If we want to reason about the execution of complex applications on MPSoCs, we certainly should not focus on modeling individual logic gates in the architecture. The challenge is to model architectures at the right level of abstraction.

In the modeling of the computation in applications, we care about the semantics of the model. It should be expressive enough to capture the application while being rigid enough to allow a compiler and system to reason about its execution and optimize it as much as possible. Hardware, on the other hand, is fixed: in software synthesis (and in this thesis) we’re not concerned with hardware design. As such, we take a more scientific role[2] to modeling hardware, as opposed to the engineering role we take for applications: We fit the model to the hardware, not the hardware to the model.

Architectures models for software synthesis have two main requirements: specification and simulation. In order to derive an efficient implementation of an application to an architecture, the model of that architecture needs to at least include the possible decisions required for that software implementation. Different PEs and their types in the architecture, scratchpad memories or Direct Memory Access (DMA) controllers, when present, are certainly necessary parts of the models. If actual physical memory addresses or concrete instructions in the ISA should also be included depends on the flow: an end-to-end compiler that produces binaries might benefit from modeling these, whereas a higher-level, source-to-source compiler might do without them if it only makes abstract decisions about resource allocation and leaves code generation to a separate compiler.

Similarly, in many cases a simulation is part of the software synthesis flow. In this case, a model of the architecture needs to allow such a simulation. Obviously, a simple analytic model requires a different level of abstraction for the architecture model than a cycle-accurate simulator. A very concrete way of considering this is the Y-chart approach proposed in [Kie+01], as depicted in Figure 2.4, which is based on Figure 6 from [Kie+01].

The Y-charts approach is closer to a co-design methodology: architectures are part of the design space, albeit only as parametrized families. As such, they model an architecture as an abstract set of parameters (e.g.

---

[2] In the sense of Lee [Lee17], as described in the introduction.

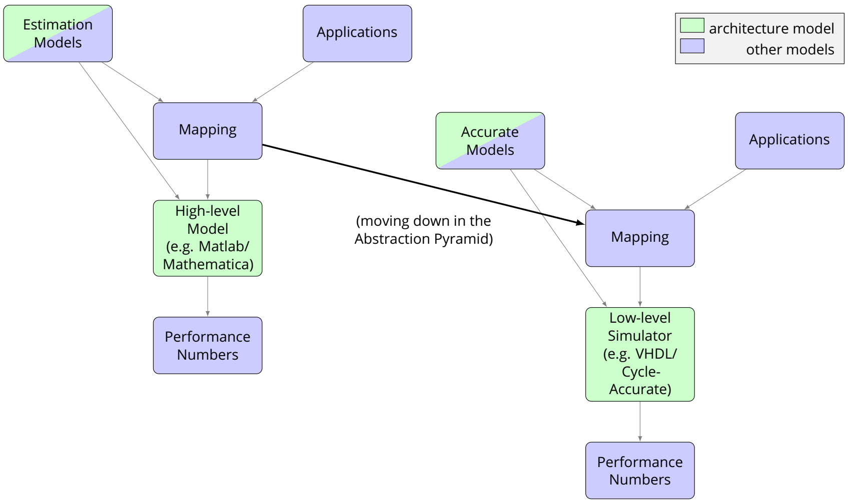


Figure 2.4: Multiple Levels of Abstraction in the Y-Chart Approach (Inspired by Figure 6 in [Kie+01]).

number of cores of specified core types) for specification (mapping), with an ad-hoc model for simulation (in matlab/mathematica) or well-defined models from a lower level of abstraction (cycle-accurate models or VHDL). Thus, the approach described in Figure 2.3 shows well how different models of architectures at different levels of abstraction can co-exist and be used. While accurate simulation is pivotal for effective software synthesis, simulation methods and accuracy are beyond the scope of this thesis. We will thus focus only on models of architecture for the sake of DSE and the specification of decisions (concretely, here, mappings).

The general situation described in the Y-charts approach is very common in practice: A parametrized family of hardware architectures is assumed as part of the flow, and architectures are described in terms of this family. With newer developments in hardware, like the proliferation of NoC-based architectures, many modern approaches apply the same principle to these modern architectures. For example, the models used by [Wei+14; Sin+10; RG18] all assume a regular mesh ($N \times M$) NoC-based topology and parametrize the architecture by the size of the mesh, N, M as well as the core types and communication and memory parameters like worst-case latency values. In the Sesame framework [PEP06], an additional abstraction layer called the *mapping layer* works as an intermediate virtual platform, in correspondence with the KPN, application, which is then mapped to the target platform. In the DOL approach [Thi+07], architectures are modeled in an XML specification that implicitly models the architectures as graphs with specific annotations e.g. for memory sizes or resource sharing methods like first in - first out (FIFO). While this is an ad-hoc model, its graph-based nature is general enough to describe arbitrary architectures. This is common of the most general models at this level of abstraction: they are graph-based models. In [ECP06], architectures are modeled as bi-partite graphs with cores and memories. This is bi-partite structure is actually similar to the constraint graphs defined in [Wei+14; RG18], which basically describe the subset of the architecture used by a mapping. In MAPS [CLA11], on the other hand, for the purposes of mapping, architectures are described by labeled graphs where only the cores are nodes and the edges represent communication. This is similar to the model described in [PEP06]. Some of these models [3] have been an influence in the SHIM standard [The15] and the IEEE 2804-2019 Standard [CDA20]. There are subtle differences between all these models, which makes comparing approaches difficult [Goe+16].

In practice, however, the different graph-based architecture models are mostly equivalent. For this thesis we use a model based on the MAPS model for defining *architecture graphs*. An architecture graph $A = (V_A, E_A, l_A)$ is a labeled directed multigraph where the nodes V_A represent PEs in the architecture. These PEs are labeled with core types. Communication in the achitecture graph is represented by the edges E_A. Since A is a multigraph, E_A is a multiset: there can be multiple edges $e_1, \dots, e_n \in E_A$ between two cores $\mathrm{PE}, \mathrm{PE}' \in V_A$. These edges are different by their label $l_A(e_i), i = 1, \dots, n$. The labels of edges identify them as *communication primitives*. Communication primitives are an abstraction that encompasses communication via multiple methods: shared memories, DMA or even specialized hardware like hardware FIFO buffers. Communication primitives can also be used to model different software libraries/APIs for communication that can use the same hardware [Ode+13].

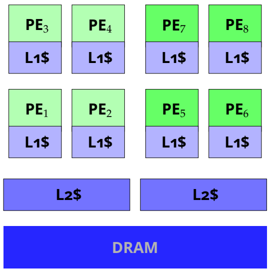


Figure 2.5: The Odroid-XU4 Architecture.

Consider the architecture depicted in Figure 2.5,the Exynos Odroid-XU4 with a Samsung Exynos 5422 chip, which has an octocore ARM big.LITTLE (4+4) architecture. This architecture has two types of cores, the ARM Cortex A7/A15 , little and big, respectively. Similarly, there are three types of communication primitives in the architecture: communication via the L1 and L2 caches, or over the shared DRAM memory. This architecture can be modeled in an architecture graph by having 8 nodes, one for each core (4 of each of the two core types), and connecting the nodes by with all the primitives that can be used to communicate between them. Figure 2.6 shows the architecture graph for this example.

In NoC-based architectures, the communication depends on the routing over the on-chip network. In particular, the communication latency changes depending on the number of hops required to communicate between two PEs. Our model of architecture graphs (among others, like the DOL architecture model) has the advantage of having a different communication primitive for each of these connections with different numbers of hops, thus being able to model NoC-topologies as well as others (e.g. BUS-based). However, for simplicity of reasoning, we can sometimes benefit of a related graph, which we call the *topology graph* [GMC18]. A topology graph $T = (V_T, E_T, l_T)$ is also a directed multigraph with the same vertex set $V_T = V_A$ as that of the architecture graph A, namely the set of cores. Thus, the labels are also identical $l_A |_{V_A} = l_T |_{V_T}$. The edges are different:

---

[3] Specifically, iterations on the MAPS model by Silexica

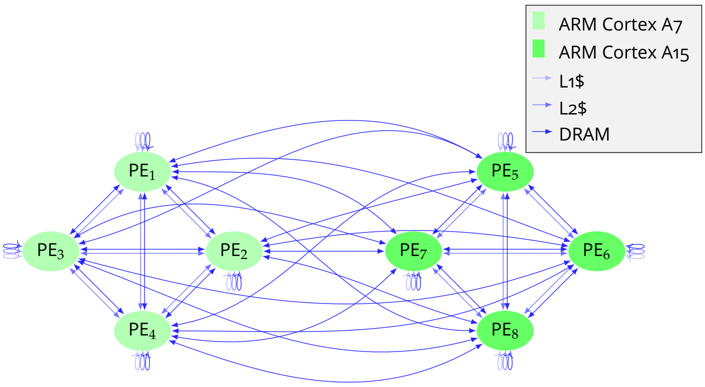
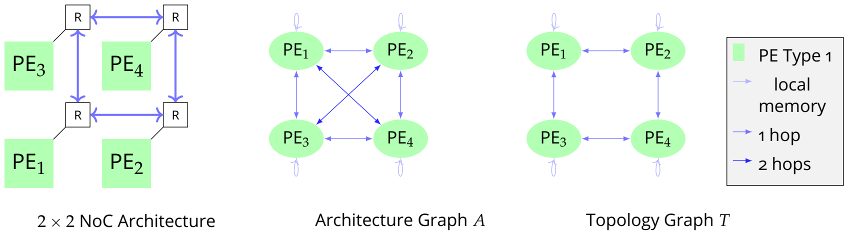


Figure 2.6: An Example of an Architecture Graph for the Odroid-XU4 Architecture.

we only add an edge for a communication primitive $e \in E_A$ if it allows **direct** communication between two cores. Thus, $E_T \subseteq E_A$. For a BUS-based architecture like the ODROID-XU4, this topology graph corresponds to the architecture graph. However, for a NoC-based architecture, the topology graph captures the network topology. Figure 2.7 shows the difference of the architecture graph A and the topology graph T for a 2 × 2 regular mesh NoC topology. The difference between the two graphs in this case is that the topology graph has no nodes for multiple hops, whereas the architecture graph has them. As such, the topology graph reflects the topology of the on-chip network better, as can be seen by comparing them in the figure.



2 × 2 NoC Architecture   Architecture Graph A Topology Graph T

Figure 2.7: Comparison of the Architecture and Topology Graphs for a 4×4-Mesh NoC-based Architecture.

As mentioned above, the subtle differences in different models make comparison between them difficult [Goe+16]. The main reason for this are the two distinct roles that architecture models play in software synthesis, as we have discussed in this section. Having a common model for specification is beneficial for defining software synthesis approaches, and thus, desirable. Having common models of architecture, while beneficial for comparison, is not necessarily desirable: there are good reasons for having simulations at different levels of accuracy. Nevertheless, Pelcat and others have [Pel+15] made an attempt to define such common models of architecture. Their definition is abstract: they require a unique, reproducible cost of computation. This solves the problem of comparability, at the cost of the simulation. In a sense, their definition of a model of architecture is tantamount to defining a specification for a simulation. We believe this is a great idea, but unfortunately not yet mature enough in terms of the models that exist and their integration to simulators. The Linear System-Level Architecture model they propose is also a graph-based model and is similar to the graph-based models discussed above. However, we believe that it is better to separate both concerns conceptually, namely simulation and the specification of mappings. As such, we will focus only on the graphs defined in this section for mapping specification and leave the simulation level open to the multiple levels of accuracy, as required by the use-case.

2.4 The Mapping Problem

The main problem we address in the first part of this thesis is the *mapping problem* [Mar+11]. The mapping problem is the decision problem of assigning physical resources (hardware) to the logical tasks and data (software) of an application. As can be seen from Figure 1.3 in the introduction, this is a central problem in software synthesis.

We commonly think of assigning the tasks and communication channels (or data) to the physical resources, and not the other way around. The reason we do not choose to do so again has a mathematical background, as we will explain here. Such an assignment is a correspondence and can be interpreted as a relation $\mathcal{R} \subseteq A \times K$ that relates the architecture A and the application K. By abuse of notation we refer to the graphs A, K here to mean both one relation on their nodes V_A, V_K and one on their edges E_A, E_K.

A relation is the most general description of such a correspondence. However, in this thesis we do not consider mappings where a single task can be assigned to multiple hardware resources. The thread affinity mechanism in the POSIX standard, for example, assigns a POSIX process to multiple (hardware) threads. Then, the operating system scheduler decides in which of the specified threads to actually execute the process, possibly migrating it multiple times during its execution. We do not consider this kind of behavior. If we want to model it with the mathematical framework proposed here, however, we can. For this, we describe the final mapping as decided by the scheduler at run-time, and consider migrations as multiple spatial mappings at different time instances.

We define a mapping to have exactly one physical resource for each logical one (i.e. for each task or data/communication channel). This kind of mathematical relation is precisely the definition of a function, which is why we model mappings as functions $m : K \to A$, i.e. assigning physical resources to the logical ones. A mapping also needs to be consistent. If it assigns two tasks $t_1, t_2 \in V_K$ to different PEs, when these tasks exchange data (i.e., $(t_1, t_2) \in E_K$), the data communication channel needs to be mapped to a physical channel that respects the task assignment: we require that $m((t_1, t_2)) = (m(t_1), m(t_2)) \in E_A$. This condition, mathematically, means precisely that a mapping respects the graph structure of K and A. In other words, a mapping is a *morphism of graphs* $m : K \to A$.

Consider the example of the mapping depicted in Figure 2.8. It shows the mapping

$$m : t_1 \mapsto \mathrm{PE}_1, t_2 \mapsto \mathrm{PE}_2, (t_1, t_2) \mapsto \mathrm{L2\$} \, .$$

This mapping can be considered as the morphism of graphs depicted on the right, where the image $m(K) \leqslant A$ is a subgraph of the architecture graph A (cf. Figure 2.6). We could not map the communication edge (t_1, t_2) to, say, the L1 cache of PE$_3$, L1\$, since this cannot be used to communicate between PE$_1$ and PE$_2$. This is equivalent to saying that L1\$ is not an edge

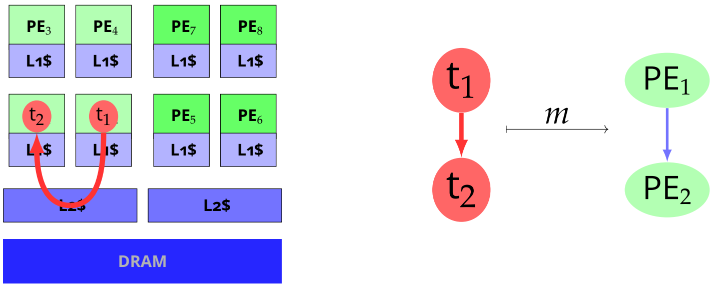


Figure 2.8: An example of a mapping as a diagram (left) and as a morphism of graphs (right).

of $m(t_1), m(t_2)$, or any L1 cache for that matter, since (more precisely) there is no edge $(m(t_1), m(t_2)) \in V_A$ with the label $l_A((m(t_1), m(t_2))) = \text{L1\$}$.

We define a set $M \subseteq \{m : K \to A, m \text{ is a morphism}\} =: \text{Mor}(K, A)$ as the set of (valid) mappings. A morphism of graphs $m : K \to A$ that is not in the set M is an invalid mapping. This might be because different reasons, e.g. if a PE $p \in V_A$ is not general purpose and cannot execute some tasks, or when modeling the sizes of data (channels), if a communication channel does not fit a physical resource. We model this by letting M be a proper subset of $\text{Mor}(K, A)$, the set of morphisms $K \to A$.

Having formally defined a mapping, we can also define the mapping problem. Let $\Theta : M \to \mathbb{R}^k_{\geqslant 0}$ be a function on the set of mappings. We call Θ an *objective function*. For example, $\Theta : M \to \mathbb{R}_{\geqslant 0}$ (for $k = 1$) can be the execution time of the application K when mapped via m to the architecture A. This could similarly be another measure of the quality of a mapping, like throughput or total energy consumption. It can also be a combination of multiple metrics for $k > 1$. Additionally, depending on the use-case, the results of the software synthesis process might need to respect some constraints. For example, we might want to minimize the energy consumption while maintaining the execution time under some real-time threshold. Let $C : M \to \mathbb{B}$ be the (boolean) function that decides if a mapping satisfies the required constraints. Thus, in the example, Θ would be the energy consumption and $C(m)$ would be true if and only if the mapping's execution respects the real-time constraint. We can generally define the mapping problem as the following multi-objective optimization problem:

$$\min_{m \in M, C(m) = \text{True}} \Theta(m) \tag{2.1}$$

Here, the minimum of the vector $\Theta(m) \in \mathbb{R}^k_{\geqslant 0}$ for $k > 1$ can be understood as an element-wise minimum. In particular, some points are incomparable: if $\Theta(m_1)_1 > \Theta(m_2)_1$ and $\Theta(m_1)_2 < \Theta(m_2)_2$, then $\Theta(m_1), \Theta(m_2)$ are incomparable. This element-wise comparison of vectors gives us a partial order on $\mathbb{R}^k_{\geqslant 0}$. Equation 2.1 can be then understood as finding Pareto-minimal points, i.e. points that are not dominated by any other point in the set. Concretely, we say that $\hat{m}$ is not dominated by any point (is Pareto minimal), if $m \nless \hat{m}$ for all $m \in M$. A variant of this same problem can be encoded as an integer optimization problem, e.g. as is done in [ECP06]. As a problem formulation, however, we believe the treatment given here defining the conditions as a morphism of graphs is much simpler to read and understand and just as expressive.

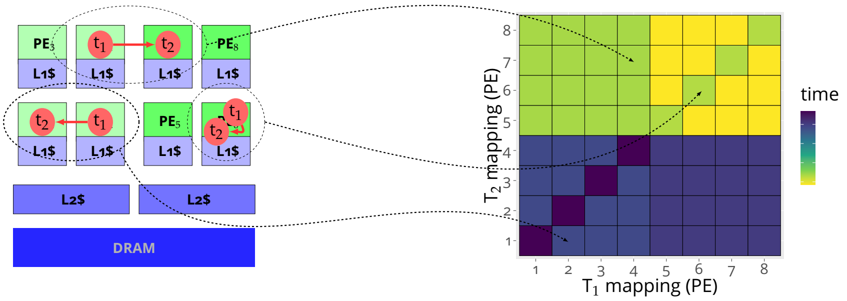


Figure 2.9: An example of the mapping space for a simple two-task application.

Figure 2.9 is a reproduction of Figure 1.4. It depicts the optimization problem of Equation 2.1 on a very simple example. The example is based on a telecom application of the E3S benchmark suite, chosen specifically because it consists of exactly two tasks, which allows the mapping space to be visualized in a two-dimensional plot. The mappings are plotted by encoding the mapping of each of the two tasks as the x and y coordinates of the grid, and the color of the squares in the grid encodes the (simulated) execution time on the Odroid-XU4 architecture. The actual values of the execution time are irrelevant here and have been deliberately omitted. In the figure it is clear that the minimal execution time is obtained by mapping the two tasks two distinct Cortex-A15 (big) cores, i.e. the set of m with $t_1 \mapsto p_1, t_2 \mapsto p_2$ where $p_1, p_2 \in \{\mathrm{PE}_5, \mathrm{PE}_6, \mathrm{PE}_7, \mathrm{PE}_8\}$ and $p_1 \neq p_2$ are precisely the minimizers of Equation 2.1.

The example in Figure 2.9 is chosen deliberately to be so simple that it can be depicted in a figure. There are exactly $2^8 = 64$ mappings in the mapping space. For the audio filter application from Section 2.1 (cf. Figure 2.1), this space has already $8^8 = 16777216$ mappings and finding the minimal execution time is much less tractable. In general, the mapping space has cardinality $|V_A|^{|V_K|}$, and thus grows exponentially with the number of tasks $|V_K|$. For the 1024-core Epiphany-V-based Parallella architecture [Olo16], the mapping space of a moderately-large application with 28 tasks has more than 10^{84} possible mappings, considerably more than there are atoms in the observable universe.

The 28-task example is by no means unrealistically large. The mapping problem is already intractable for problem sizes in practice. As such, multiple clever algorithms and heuristics have been designed with different settings, objectives, and assumptions. A survey of these mapping approaches can be found in [Sin+13]. The focus of this thesis are not mapping heuristics, but rather the mapping problem itself and general structural properties that can be exploited in these heuristics. As such, we will not dive deeply into the literature of heuristics. At this point, we will only distinguish between general classes of mappings: static, dynamic and hybrid, as well as between heuristics and meta-heuristics.

Static mappings are mappings defined prior to run-time (commonly, at design-time or at compile-time). Traditionally, mapping decisions made at design-time are part of a co-design approach and are hard-coded into the application itself. By abstracting over the target architecture and the mapping problem, software synthesis methods allow to defer these static decisions to compile-time, and make this kind of mappings more portable.

Dynamic mappings, on the other hand, are chosen at run-time. The dynamic mapping problem is in essence the same as task scheduling. The trade-offs between time available to make a (scheduling) decision and the available information at run-time are certainly not unique to the mapping problem. However, dynamic mappings present an additional hurdle in heterogeneous systems, since code has to be compiled for the different possible targets.

Hybrid mapping approaches sit between static and dynamic ones. An ahead-of-time decision process or mapping space pruning analyzes the mapping space and pre-defines a set of mappings or partial mappings. From these pre-defined mappings, a run-time system chooses a mapping or constructs a mapping from the partial mappings, based on the available information at run-time.

Finally, we distinguish between heuristics and meta-heuristics. Mapping heuristics, like load-balancing, are domain-specific algorithms that exploit the specific domain-knowledge to find a solution based on a pre-defined model of the problem. On the other hand, meta-heuristics, like genetic algorithms, rely on an iterative evaluation of the points. In the case of mappings, this usually means a simulation or profiling of a mapping's execution. Again, this distinction is not unique to the mapping problem.

2.5 Simulating Mappings

Simulations are extremely important for analyzing an application's performance, or more generally, its behavior. As described in Section 2.3, there are multiple levels of detail in which to model and, consequently, simulate, an architecture and its execution. For investigating the mapping problem in software synthesis, higher-level simulations are preferable for multiple reasons. First and foremost, higher-level simulations are faster. If a meta-heuristic iteratively evaluates dozens, hundreds or even thousands of mappings to find a near-optimal one, it greatly benefits from the fast evaluation time associated with a higher-level simulation.

Higher levels of abstraction come with a trade-off. The accuracy of the simulation suffers in exchange for the simpler models and faster simulation times. Let $\tilde{\Theta}$ be the approximation of Θ from the simulation. A loss in accuracy means that $|\Theta(m) - \tilde{\Theta}(m)|$ becomes larger. However, depending on the use-case and mapping objective Θ, this loss in simulation accuracy might not necessarily affect the quality software synthesis results. Suppose that the objective Θ represents execution time or energy consumption, and the goal of the software synthesis is just a best-effort minimization of Θ (with no additional constraints, i.e. $C \equiv \mathrm{True}$). Then the accuracy of the simulation is not important, only its fidelity. If $\Theta(m_1) < \Theta(m_2)$ we want the result of the simulation to reflect this, $\tilde{\Theta}(m_1) < \tilde{\Theta}(m_2)$. As long as this is the case, we don't care about the actual value of $|\Theta(m_i) - \tilde{\Theta}(m_i)|$, since in this case the exploration will still find the minimum. The fidelity of the simulation is a measure of how often this is true. On the other hand, if the application is a real-time application, then the truth value of C will depend on the accuracy of the simulation. Here, the accuracy of the simulation is much more important.

This chapter describes the simulation aspects which pertain the models of computation and the practical tooling we will use. Nuanced simulation details and advanced techniques are beyond the scope of this thesis.

2.5.1 *Simulating the Execution of Kahn Process Networks*

The behavior of a system plays a central role in simulation. A deterministic model should yield deterministic simulation results. Non-determinism, when present, should also be captured by the models and reflected by the simulations.

The behavior of systems is commonly captured in execution traces, which simply record the behavior of different entities (e.g. processes or actors) at different timepoints. This can be formally captured in a monoid structure of (Mazurkiewicz) traces or, equivalently, histories [DR95], as described in Section 2.2. Traces are common in many domains, as they useful to understand the behavior of systems [Nag+96]. However, for systems that are non-deterministic, (by definition) the behavior of the system does not only depend on the input. This can make designing [Lee06] and debugging [Mur+14] particularly difficult. In cyber-physical-sytems or, more generally, reactive systems in the sense of Harel and Pnueli [HP85], input from the physical world might come in a non-deterministic fashion. The problem of capturing the behavior of such a system is even more complex when the system is distributed [Sha16].

Kahn Process Networks are deterministic, as are all the dataflow models that can be embedded as KPNs. This means that the behavior of a KPN application depends only on the input to the network. In particular, it does not on the mapping and scheduling or related execution details. Thus, their behavior can be captured by a (Mazurkiewicz) trace. This permits to re-create their behavior in a fashion that is independent of the mapping. By "replaying" the trace, i.e. simulating the execution of a process for every input in the trace, a discrete-event simulator can successfully simulate the execution of a KPN, since the token sequence is guaranteed to be identical given identical inputs. In particular, this allows us to do Design-Space Exploration (DSE).

A discrete-event simulation of a KPN application thus requires behavior traces. It also needs to model the execution and communication times. Modeling execution times from a trace is simple, with a crucial assumption: if the execution times for a trace event only depend on the PE type. This assumption will not always hold, e.g. when the instruction cache is flushed due to scheduling decisions, or due to unpredictability from the operating system (OS). Note that data caches are modeled as part of the communication between processes. In most cases this assumption is a good approximation, as it is normal to expect that the same code executing on the same data and the same ISA will usually require the same amount of time.

Modeling communication is more complicated, as it depends on the memory subsystem. In general, the communication costs of sending a KPN token depend on multiple factors, like the size of the token, contention in the memory subsystem (and correspondingly methods of arbitration, routing in the case of a NoC, etc), or the API and protocol being used. For the simulations in this thesis, we use a model based on annotations of the architecture graph A. These annotations are functions that calculate the time cost of communicating data, as a function of its size. In this way, we model both the latency and bandwidth of the communication. We use a split-cost communication model to assign costs to sending and receiving data [Ode+13]. This separation can be used to simulate based on traces, as described above, since we can then compute the cost of communication for both the sending and receiving processes.

When dealing with NoC-based architectures, this model is not as accurate. Communication over a NoC depends also on the routers and links along the path, including the routing algorithms. We extended the split-cost communication model to account for these issues in [MGC16]. The idea is to add a third term to account for the network, in addition to the consumer and producer costs. This third term can account for the routing and the topology of the network while maintaining an analytic model which is cheap to evaluate in a high-level simulation.

Simulation is essential for software synthesis, yet it is not the focus of this thesis. The main contribution of [MGC16], with a concrete model for the Tomahawk 2 architecture [Noe+14] and the corresponding evaluation comparing to the SystemC-based simulator Noxim [Cat+15] are due to my coauthors and beyond the scope of this thesis.

2.6 Software Synthesis Flows

Many flows exist that enable model-based design in a software synthesis flow. In the introduction we discussed some of the original software synthesis methods, the approach of [Lin98] uses Petri Nets, or [RPM92] which uses SDFs and other flows which use multiple models [BLM00; Pin+95; BML12], generally dataflow.

SystemCoDesigner [Hau+08] is based on SystemC and aimed at FPGAs, as is the case with CAPH [SBA13], which is based on dataflow and the actor model. Although the flows are based on MoCs, their goal is not software but rather an FPGAs implementation, and as such these flows are closer to HLS than the rest. Coincidentally, the term software synthesis is an allusion to the much better-known HLS.

Also based on a more general dataflow model is the Turnus [Cas+13] flow. It builds on top of RVC-CAL, which is in turn based on the CAL actor language [EJ03].

More specific is the SDF For Free (SDF^3) [SGB06] framework, which does much more than generating random SDF graphs. As a software synthesis tool [SGB10], it focuses on the more restricted SDF MoC, allowing much more sophisticated analysis of the applications. Similarly, PREESM [Pel+14] works with parametrized extensions of SDF [Des+13] that provide a greater trade-off between expressiveness and analyzability.

On the other side of the MoC spectrum, many related flows use KPN. The static-mapping-based flows of Distributed Operation Layer (DOL)[Thi+07], Sesame [Erb+07] or MAPS [CLA11] use different levels of abstraction to derive an efficient execution form a KPN-based application description.

Going beyond static mapping, the DAARM [Wei+14] flow maps dataflow applications using a hybrid approach. Similarly, the work of [QP15] extends the Sesame approach to hybrid mappings, and Spider [Heu+14] extends the work of PREESM to hybrid mappings.

This thesis and the contributions included in it are **not** aimed at proposing (yet another) software synthesis design flow. Instead, we propose methods to improve existing flows, with the ambitious goal of being general enough that the improvements would benefit most of the flows discussed. Perhaps a good way to think of this is: Just as these flows help users write more efficient applications, we aim to help the flow designers improve their flows. For chapters 4 and 5, and partially Chapter 3, we focus on one flow to do this. We choose the MAPS flow [CLA11], which we describe next. Some contributions, on the other hand, go beyond these flows. This is particularly the case in chapters 6 and 7, and in part Chapter 3

2.6.1 *The MAPS flow*

The MPSoC Application Programming Studio (MAPS) is a software synthesis flow developed at RWTH Aachen University and spun-off into a company, Silexica[4], which kindly allowed us to use the KPN mapping flow of MAPS for our research. MAPS is very comprehensive, it does much more than KPN-based software synthesis. It has analysis algorithms to suggest parallelization of sequential code, both as OpenMP annotations as well as CPN annotations. We will not discuss these here. It also has detailed platform models which are used in simulation and performance estimation flows, we will very briefly outline these, only as they pertain the KPN-based software synthesis flow.

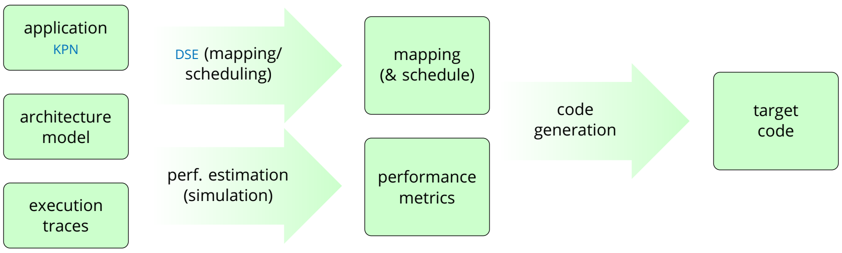


Figure 2.10: The Software Synthesis Flow from Figure 1.3. MAPS implements all steps in the flow, which are therefore all depicted in green.

Figure 2.10 describes the MAPS flow, as an instance of the general software synthesis flow in Figure 1.3 from the introduction. Applications are written as KPN applications in the CPN language. While CPN supports SDF annotations as well, these are embedded into the KPN MoC for analysis and code generation, there is no separate DSE and code generation for purely-SDF applications. The architecture model is an XML-based description which has detailed models of the communication subsystem and its topology, including different possible communication APIs [Ode+13], different frequency and voltage domains and even models of the ISA for the processing elements. This model influenced the definition of the SHIM standard [The15] which then resulted in an IEEE standard [CDA20].

Performance estimation in MAPS follows in multiple steps. In a first step, using a POSIX threads (pthreads) backend, the application is emulated on the host machine to gather functional KPN traces. Since the KPN model is deterministic, these traces are independent of the actual performance values of the application. Then, the processes are instrumented and executed in isolation, dividing them into what in MAPS are called *segments*. These segments are defined as the execution between any two reads or writes to or from channels[5]. Using the data from the functional KPN trace, MAPS obtains a detailed trace of the instructions executed during each segment in the process. These detailed instruction traces are combined with an abstract processor model from the architecture description

---

[4] `https://www.silexica.com/`

[5] A special annotation can be used additionally to divide segments manually.

to estimate the performance on the target platform [Eus+14]. This yields traces with performance annotations for every process and every PE type. Finally, these performance-annotated traces are used in conjuction with the mapping and communication model in a discrete-event simulator to estimate the overall performance of the mapping. If performance-annotated traces are available from a profiling execution on the actual hardware, these can be used instead.

The DSE step in MAPS is similar to that of all the flows described at the beginning of this section, as well as the flow of the `mocasin` tool, which we will describe shortly. In this DSE step, MAPS generates a mapping. On some platforms, when processes share a PE, MAPS can also generate a schedule for the processes. These mappings are then used by the Clang-based CPN compiler to generate target-specific C code, which can be further compiled by a C compiler for the target platform. This way, the flow generates target-specific code from an abstract KPN description of the application (and the appropriate platform models).

As explained in the introduction, this thesis does not focus on the performance estimation and code generation steps of this flow (cf. Figure 1.3). We use MAPS for performance estimation and code generation. For evaluating our methods in DSE and application, architecture and mapping models, we primarily use the `mocasin` tool, which we describe in the next section.

2.7 The `mocasin` tool

In this thesis we will use `mocasin`, an open-source [6] tool for the MoC-based analysis and simulation of applications [Men+21]. This tool, formerly known as `pykpn` [MGC16; GMC18], has been developed as part of a collaborative effort between multiple researchers at the Chair for Compiler Construction at TU Dresden. While the tool itself is a joint contribution with the coauthors of [Men+21], many concepts introduced in this thesis have been implemented and tested using `mocasin`. As such, this section will explain the tool in-depth, to enable the description of the different implementations of contributions from this thesis implemented in `mocasin`.

Figure 2.11 depicts the basic flow of `mocasin`, which can be understood as a tool for rapid prototyping of prototyping tools. Multiple dataflow MoCs are supported by `mocasin`, like SDF or task graphs. These models, among others, can be seen as specializations of KPN [LP95] and will be discussed more in-depth in Chapter 6. The `mocasin` architecture is composed of multiple modules that can be combined to create a specific tool (e.g. for mapping or simulation). In the figure we show the modules that are relevant for this thesis.

In general, simulating a KPN requires four inputs, as explained in Section 2.5: the KPN graph, a platform description, execution traces and a mapping. The `mocasin` tool has data internal structures for these four inputs that reflect the models as explained insection 2.1-2.4. The tool boasts multiple readers to generate the internal data structures from established formats like `tgff` [DRW98], `sdf3` [SGB06] or the `MAPS` formats. Instead of a concrete trace, `mocasin` expects a trace generator, which can generate the trace on the fly: this is useful e.g. for non-deterministic models computation, but for KPNs the two are equivalent. The KPN trace

---

[6] https://github.com/tud-ccc/mocasin

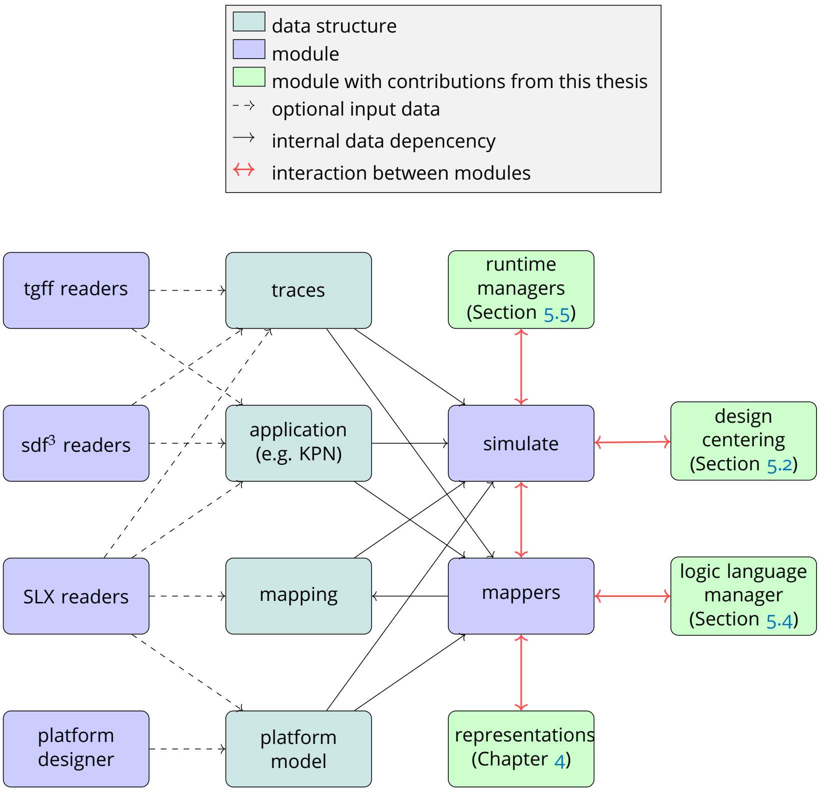


Figure 2.11: Mapping and simulating KPN Applications in `mocasin`.

generator, for example, simply reads the trace from a file. A mapping, while required for simulation, does not need to be provided: it can be calculated in a Design-Space Exploration. This is not surprising, since a significant part of this thesis concerns itself with improving such mapping algorithms.

A central part of `mocasin` is a discrete-event simulator [MGC16] that uses the principles outlined above to simulate KPN applications based on their traces (as well as other models of computation). We will not dwell on the design of the `simulate` module since it goes beyond the contribution and scope of this thesis. Many contributions of this thesis are implemented in `mocasin`. This is done as modules, using the `mocasin` toolbox infrastructure. Different contributions of this thesis and the corresponding references are described in the figure and marked as such (with light-green coloring). In the following, we will describe some other central modules of `mocasin`.

platform designer

Many concepts developed in this thesis are aimed at emerging technologies and future hardware architectures. To model these increasingly complex architectures, we aim at an abstract description of their topologies (cf. Section 2.3). As part of `mocasin`, with the help of Felix Teweleitt, we designed a modeling infrastructure, in essence a small embedded domain-specific language, to describe hardware topologies. This infrastructure is the `platform_designer` module of `mocasin`.

Listing 3 shows an example of our `platform_designer`. The code in this listing describes the topology of the Odroid XU4 (see Figure 2.5). The main

```
pd = PlatformDesigner(self)
pd.setSchedulingPolicy('FIFO', 1000)
pd.newElement("Odroid-XU4")
# cluster 0 with l2 cache
pd.addPeClusterForProcessor("cluster_a7", processor_0,
                            num_little)
# Add L1/L2 caches
pd.addCacheForPEs("cluster_a7", 1, 0, 8.0, float('inf'),
                  frequencyDomain=1400000000.0, name='L1_A7')
pd.addCommunicationResource("L2_A7", ["cluster_a7"], 250, 250,
                            float('inf'), float('inf'),
                            frequencyDomain=1400000000.0)
# cluster 1, with l2 cache
pd.addPeClusterForProcessor("cluster_a15", processor_1, num_big)
# Add L1/L2 caches
pd.addCacheForPEs("cluster_a15", 1, 4, 8.0, 8.0,
                  frequencyDomain=2000000000.0, name='L1_A15')
pd.addCommunicationResource("L2_A15", ["cluster_a15"], 250, 250,
                            float('inf'), float('inf'),
                            frequencyDomain=2000000000.0)
# RAM connecting all clusters
pd.addCommunicationResource("DRAM",
                           ["cluster_a7", "cluster_a15"], 120,
                           120, 8.0, 8.0,
                           frequencyDomain=933000000.0)
pd.finishElement()
```

Listing 3: The Odroid-XU4 Platform with the Platform Designer principal innovation behind the `platform_designer` is that it works with a stack of clusters. The functions `newElement()` and `finishElement()` can be nested to describe the topology in a hierarchical fashion. Between these functions, the API allows us to describe heterogeneous cores and different levels of interconnects with different properties, like their frequency.

mappers

The mapping problem (cf. Section 2.4) plays an important role in this thesis. While we do propose some mapping heuristics for special contexts, many methods in this thesis are orthogonal to the mapping heuristics. As part of this thesis we have implemented multiple mapping algorithms from literature in `mocasin`. These can be found in the `mapper` module. The heursitics included are the Group Based Mapping (GBM) heuristic [Cas+12] and a static variant of the Linux Completely Fair Scheduler (CFS). We also have some meta-heuristics, which include a random walk, simulated annealing [Ors+07], tabu search [MEP08] and genetic algorithms [ECP06].

configuration

`mocasin` is designed to be a tool for tool development. As such, one of its main goals is to enable building different scenarios for different contexts, like static mapping of KPN applications or hybrid execution of dynamic Long Term Evolution (LTE) loads [Men+21]. We use the Hydra [Yad19] framework to configure `mocasin` and construct different scenarios as different tools. This configuration philosophy allows us to work in a modular fashion, which in turn allows us to implement different contributions of this thesis as `mocasin` modules.

3

BENCHMARKING

The methods we will discuss in this thesis are ultimately about improving the performance of software. We need benchmarks to assess the performance of software, and consequently to assess if the performance improves. Benchmarks are essential for the research and development of compilers and programming languages [HPP09], as well as hardware architectures or runtime systems. In this context, benchmarks are generally understood to be collections of programs with particular properties. Mostly, they cover a range of behaviors that are typical of, and important for programs in a particular domain. This description, however, can mean several different things. In this chapter, we formally define different types of benchmarks and use them to classify different use cases. We then proceed to discuss concrete KPN and task-graph benchmarks for software synthesis, as well as benchmark generation strategies both with random graph models and machine learning.

3.1 Representative Benchmarks

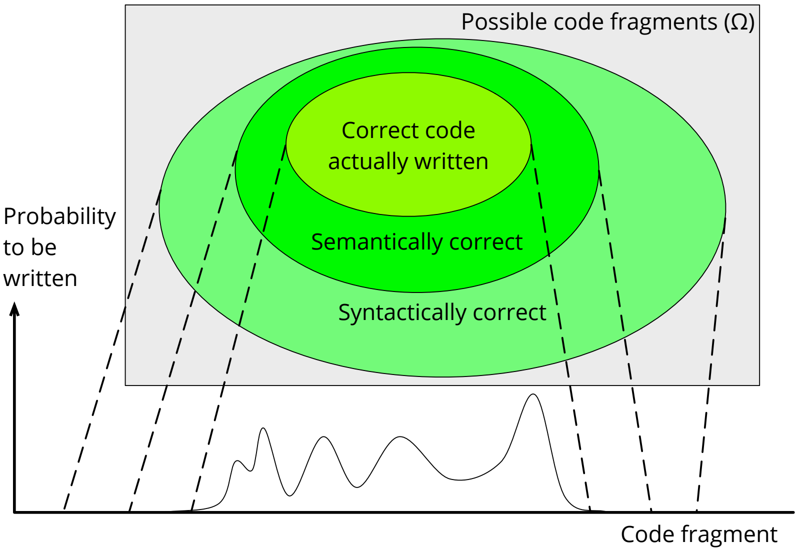


Figure 3.1: An illustration of probabilities in code space

To formalize our argumentation, we take a statistical view of program code. Consider a formal language that describes the set of all possible programs. For a program of fixed bounded size, this is a finite set Ω. For example, the set of syntactically correct C source files smaller than 1 TiB in size is certainly bounded by $|\Omega| \leqslant 2^{2^{40}}$. Out of the syntactically-correct programs, only a fraction successfully compiles, and an even smaller fraction executes something that makes sense semantically. Ideally, code written by developers falls into the subset of executable programs, as an even smaller subset. However, in this subset of correctly written code, not every code fragment is going to be equally common. A fragment like `for(int` `i = 0; i < n; i++)` is probably going to be seen much more frequently than something like `(*(&main+0x134))(x)`. It is worth noting that there is a more nuanced discussion behind what constitutes a unit of code. At this point, however, we can omit this discussion and consider the whole program as a unit, for simplicity of the argumentation. Thus, there is an implicit probability density function (pdf) p on the discrete set of possible code units Ω which models the way programmers write code. This is depicted in Figure 3.1. In reality, this is a highly dimensional space, and many challenges would arise in defining a proper geometry in such a space. We depict the code space Ω as one-dimensional for illustrative purposes, just like the continuity of the pdf, which we have no reason to assume.

In this statistical view of code, we can consider some precise questions: What does it mean for a collection of programs to be a benchmark? More precisely, what does it mean for it to be representative, or what properties would be desirable of such a collection of programs? Consider the examples depicted in Figure 3.2. This figure depicts histograms for three kinds of collections of programs along the (implicit) pdf described in Figure 3.1. We think of an idealized abscissa dimension, with a proper metric, such that programs that are semantically close are close on this dimension. Obviously, a multi-dimensional formalization would be better for this, but we stick to a single dimension for the intuition provided by the figures. Thus, a bin in the histogram might contain a single program but represent a large category of mostly very closely related programs.

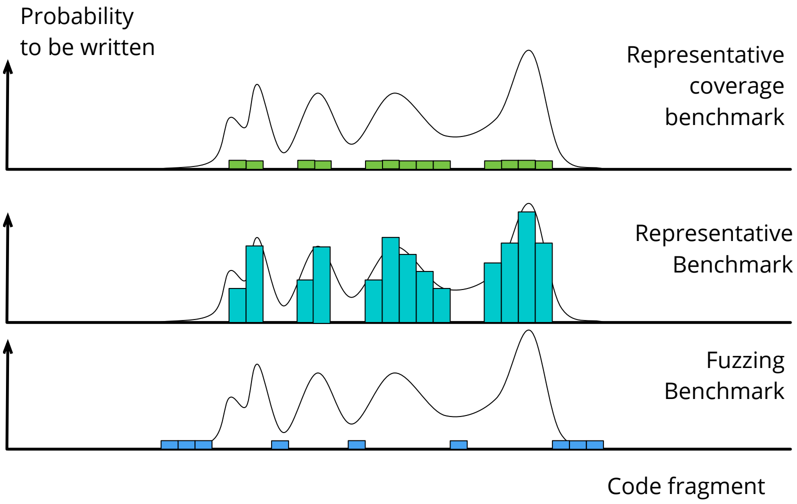


Figure 3.2: An illustration of different types of benchmarks

The first kind of collection depicted, labeled as a “representative coverage benchmark”, has a handful of programs, each of which correspond to a different category in the space of probable programs. Programs that could be written by a human, but where it is unlikely that this will happen, are not covered by this type of benchmark. Furthermore, for every type or category of code fragments, there is only one representative example in the set. In particular, programs that are moderately likely will be represented just as much as programs that are extremely likely to be written. This, in a sense, overrepresents the former and underrepresents the latter.

The second kind depicted, labeled as “representative benchmark”, removes this imbalance. It is similar to the “representative coverage benchmark”, but the difference is that in this kind of collection, programs appear with a relative frequency that is roughly in line with their probability to be written. A benchmark of this kind would probably have more programs than a “representative coverage benchmark”, without including significantly more types of programs or behaviors.

Finally, a “fuzzing benchmark” is a collection that does the opposite of a “representative coverage benchmark”. It has programs covering those programs that are unlikely to be written by a human, but possible: The corner cases.

3.1.1 *Sample use cases*

We argue that what kind of benchmark is most appropriate depends on the use case. To illustrate this, we will explain two large classes of usecases that require benchmarks. This certainly does not constitute an exhaustive classification, but will hopefully help clarify how the benchmark choice is nuanced.

Testing

A very common use case for benchmarking is testing. Assume we have developed a compiler optimization[1] and want to see how good it works. For this, we want to find out, in case someone writes a program and tries our optimization on it, how we can expect it to behave. More formally, we have a property $\mathcal{P}$ of code, like the speedup obtained by applying our compiler optimization. We want to calculate the expected value $E[\mathcal{P}]$ over the implicit pdf of writing the code we use our compiler on[2].

For testing, we argue that we want a *representative* benchmark. Ideally, we would get a set of programs $x_1, \dots, x_l \sim p$ i.i.d., where p is the implicit pdf of code been written[3]. The expected value $E[\mathcal{P}]$ can thus be approximated arbitrarily well with growing sample size l. We do this because, in our example, we assume that the users of our compiler will also draw from this distribution p, and thus $E[\text{speedup}]$ tells us what speedup the users can expect to get out of our optimization.

If we use a “representative coverage benchmark”, we can get a skewed result, because of the over- and underrepresentation of program types in this kind of benchmark. Thus, if our optimization works extremely well for a small class of programs with a moderate chance of occurring, and not so well with the most common types of programs, our testing would return wrong results. It would tell us that our optimization is likeley to improve our program, by overshooting the weight given to the moderately common class where it serves well. In practice, however, our optimization would be unlikeley to bring much improvement in this case, if we expect our compiler to be used by everyone.

---

[1] A good mapping heuristic in software synthesis can be considered a compiler optimization in this context.

[2] Technically, using the compiler is a conditional clause on the probability of a piece of code to be written by a human.

[3] A compelling case can be made that in some cases it’s the “dynamic” property of the probability that a piece of code will be *executed*, not necessarily written, that is most interesting here.

Tuning a Heuristic

Another common use case is tuning a heuristic. Consider again a compiler optimization as an example. In this case, however, instead of having a finished optimization that we want to test, we are designing the optimization by tuning a heuristic that is part of it. We want the heuristic to be tuned such that the optimization works best (which we would asses e.g. by testing, the other use-case). Training machine learning models also falls under this category, and is thus likely that this use case will continue to increase in its importance in the feature.

For tuning the heuristic, an argument can be made for all three kinds of benchmarks from Figure 3.2. It depends on the heuristic. Assume we're dealing with a code transformation (e.g. converting Python 2 code automatically into Python 3), which either it works or it doesn't. We want to optimize the parameters of our heuristic so that it works on the most cases possible. In this case we probably want a "fuzzing benchmark", to be sure we cover the corner cases, or better yet, a combination of a "fuzzing benchmark" and a "representative coverage benchmark". On the other hand, if the heuristic is something like a transformation expected to speed up the execution, then the argument for a "representative benchmark" is basically the same as for testing. We want it maximize the expected value of this speedup. An important distinction between heuristics pertains the way the parameters are set. Depending on how they are updated, repeatedly seeing similar code examples might be useless or even counter-productive, such that a "representative coverage benchmark" might be best suited.

More importantly yet is the process of designing the heuristic, before it is tuned. Usually this process is iterative. In it, having to look at the corner cases is common, too. Arguments for all the discussed kinds of benchmarks can thus be made in similar fashion for the process of designing a heuristic, depending on specific goals. For our methods improving software synthesis, we mostly want "representative coverage benchmarks".

In [Goe+19] we systematically classified all benchmarks and their usage in papers in the CGO and PACT conferences between 2013 and 2016. Table 3 in that work shows the analysis of 20 research papers from the conferences and years mentioned and the benchmarks used, metrics evaluated and classification for benchmark type. In particular, the analysis shows that most papers aim to characterize some improvement and require what we here call a "representative coverage benchmark". A few papers also used benchmarks as input for training or tuning a heuristic.

3.2 KPN Benchmarks

The `mocasin` framework supports three input formats at the time of this writing: `tgff`, `MAPS` and `sdf3`. We will discuss the first two for benchmarking here, while the `sdf3` format will be discussed in Section 3.3.

3.2.1 *CPN Benchmarks*

The first input format for `mocasin` is the `MAPS` format, which uses benchmarks written in the CPN language (cf. Section 2.1). A CPN application is compiled using the MAPS flow, which evolved into the commercial tool suite from SLX. SLX generously provided access to their tool suite for this benchmarking, as well as some CPN benchmarks.

The tool flow compiles an application using a pthreads back-end with instrumentation to record all tokens read and written in a trace. Since this traces are deterministic and depend only on the inputs, not on the execution order, they can be used to replay and simulate the execution (cf. Section 2.5). In an instrumented run, the MAPS flow also executes each process in isolation (with the stored tokens), gathering information about the precise instructions executed. This is used for performance estimation using an abstract processor model[Eus+14]. The performance estimation for each process, together with the execution traces can then be used to simulate a mapping with `mocasin`, as explained in Section 2.5.

We use three CPN benchmarks for evaluation. The first benchmark is the audio filter example as seen in Chapter 2, which takes a stereo file and implements a low-pass filter after transforming to the frequency, and transforming it back afterwards. This benchmark has 8 processes and 15 channels (some channels, used to send parameters, are not depicted in Figure 2.1). The second benchmark we consider is an embedded pedestrian-recognition application using an algorithm based on the Histogram of Oriented Gradients (HOG) technique. This benchmark was kindly provided by SLX and consists of 10 processes with 33 channels. Finally, we use a speaker recognition application, as described in [BCJ19; BJC21]. Figure 3.3 shows the graph of the speaker recognition application. The speaker recognition application has 12 processes and 33 channels.

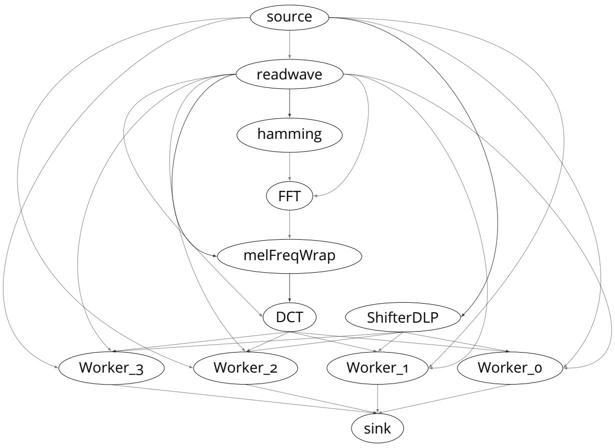


Figure 3.3: The KPN graph of the speaker recognition application.

3.2.2 *The E3S Benchmarks*

A second input of `mocasin` we use for benchmarking is `tgff`. The `tgff` format comes from task graph for free (TGFF), a random task graph generator [DRW98]. The same author also published a benchmark suite, the Embedded System Synthesis Benchmarks Suite (E3S) [Dic08]. Based on data from the Embedded Microprocessor Benchmark Consortium, the suite provides task graphs and processor execution times for applications from multiple embedded domains. In total, the E3S boasts 20 benchmarks from 5 different domains, with up to 9 tasks per benchmark. Table 3.1 summarizes the applications.

Table 3.1: Summary of applications in the E3S

Domain	No. of task graphs	tasks per graph
auto-indust.	4	4-9
networking	4	1-4
telecom	9	2-6
consumer	2	5-7
office-automation	1	5

The benchmark suite is pretty dated, being over 20 years old at the time of this writing. Unfortunately, benchmarks are generally scarce. The methods investigated in this thesis here have more to do with the trends than the actual numbers, which is why using such a dated benchmark suite is still adequate. We expect the relative performance of mapping algorithms on the E3S benchmarks to be similar to that on present and future applications, since the importance is the interplay between communication and computation costs, not the absolute values thereof.

A significant focus of the methods we will evaluate with these benchmarks is on the multicore architectures. For this, we use the same method as in [Wei+14; Sch+17]. We use the architecture topology of a modern multicore, including the frequencies, as well as the memory subsystem with its latency and bandwidth, and scale the numbers from the E3S for each of the cores of the modern multicore. This gives a realistic scenario, albeit not simulating a concrete instance of the architecture. In [Sch+17] the authors do this to create architectures with a regular mesh structure, with less realistic topologies like heterogeneous meshes with randomly placed cores. Instead, we use the topologies from concretely proposed or existing systems like the HAEC [Fet+19] or the Kalray MPPA3 Coolidge [inc20] and map the processors in these architectures to those in the benchmark suite.

3.3 Random Benchmarks and Level Graphs

The third category of inputs for `mocasin` is `sdf3`, which uses the SDF^3 framework [SGB06]. This framework is based on TGFF, adapted to the SDF model of computation. We will discuss SDF more in detail in Chapter 6. However, for the purposes of benchmarking as discussed here, both SDF and task graphs can be considered as special cases of KPN. The random graph generation of the SDF^3 framework allows multiple configurations on the types of graphs it generates, controlling the number of actors (processes) as well as the degree of connectivity in the graph, firing rates and execution times of the actors, or if the graph is acyclic.

Random benchmark generation has two main advantages over using fixed benchmarks. The first advantage is the amount of benchmarks, which is virtually unlimited with a random generation approach. The second advantage is the control over the properties of the benchmarks. Using SDF^3 we can consider precisely what effect the properties of the graph have on the algorithms (e.g. its size, or connectivity), by generating benchmarks which have the desired parameters for the independent variable we are investigating. The main disadvantage is obvious: random benchmarks are not as realistic as actual benchmarks. It is not clear if we will find a graph like the one generated by SDF^3 in a real-life application.

Since we have both the CPN and the E3S benchmarks, we will focus our evaluation on those. Instead of discussing the graph generation in SDF^3, we will discuss random benchmark generation from a different type of graph, *level graphs*[Goe+18]. The main difference is that for the use-case for level graphs in [Goe+18] we *do not* have better, realistic benchmarks we can use instead.

The context for benchmark generation we will discuss here are micro-service-oriented architectures. Large internet companies like Facebook or Twitter have an infrastructure that consists of multiple micro-services that depend on each other [Mar+14]. A crucial factor for optimal performance is the amount of I/O calls these micro-services make. We will discuss the use case more in-depth in Chapter 7. In this section we will only focus on the benchmark generation.

The micro-service-based infrastructures from large companies like Facebook or Twitter are the intellectual property (IP) of these companies and not in the public domain. If, for example, we want to improve a method for optimizing I/O in Facebook's spam-fighting service [Mar+14], we cannot use a large representative benchmark sample from Facebook to test against their method. Instead, we observe the general structure of the programs in their work and device a methodology for generating random benchmarks, with a method we call level graphs [Goe+18].

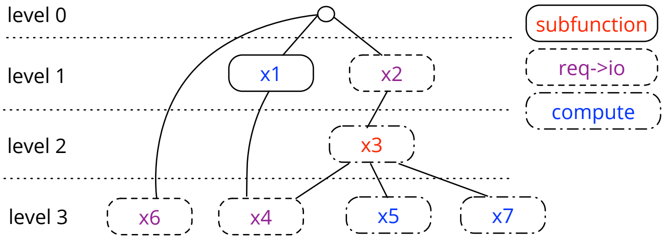


Figure 3.4: An example of a Level Graph. Adapted from Figure 1 of [Goe+18].

Figure 3.4 shows an example of a level graph. The graph depicted is a tree which is organized by levels, which are indexed with integer numbers. The nodes in the graph are labeled as different kinds of node, namely `req→io`,`subfunction` and `compute`. The graph depicted in Figure 3.4 is designed to benchmark I/O optimization, which is why the node labels are designed accordingly, reflecting I/O calls and other computation, as well as an additional `subfunction` node that creates nested benchmarks with additional function calls. This is also by design, to test the use-case.

The idea behind level graphs is to reflect the intuition of *locality* in code. This intuition is based on the observation that long-range dependencies in code are less common than short-ranged ones. While programmers do sometimes refer back to identifiers defined far behind, it is far more common to define values before using them. We interpret this as a statistical feature of the distribution of code as commonly written by humans (cf. Section 3.1). Levels in level graphs are thus designed to define the probability distribution of dependencies in graphs.

There are generally two accepted models of random graphs, the Erdős-Réyni approach [ER59] and the Gilbert approach [Gil59]. The former defines a uniform distribution over all graphs for a given number of nodes, while the latter defines the probabilities of the edges independently. Our definition of *Level Graphs* is based on the Gilbert approach, but instead of having uniform probabilities, the probabilities are defined through the levels. Concretely, a level graph $L = ((V, E), l)$ is a directed graph (V, E) with a level function $l : V \to \mathbb{N}$ to the natural numbers, with the property that for all nodes $v, w \in V$ there can only be an edge $(v, w) \in E$ if the level of v is smaller than that of w, i.e. $(v, w) \in E \Rightarrow l(v) < l(w)$. To generate a probability distribution in level graphs we define the probability of the edge $(v, w) \in E$ to be as follows:

$$p((v, w)) = \begin{cases} 0, & \text{if } l(v) \geqslant l(w), \\ 2^{l(v)-l(w)}, & \text{otherwise.} \end{cases}$$

The method can be generalized by choosing a different probability for the case where $l(v) < l(w)$. The chosen value $2^{l(v)-l(w)}$ is, to an extent, arbitrary. This probability definition ensures that dependencies are more common locally, between levels that are close by, discouraging but not prohibiting long-range dependencies.

A level graph can be used to generate code in different languages or back-ends, expressing the same computation. In [Goe+18] we implemented three back-ends for I/O optimizing frameworks, one for Ÿauhau [Ert+18] (see Section 7.3), one for Twitter's Muse [Kac15] and one for Facebook's Haxl [Mar+14]. These back-ends are also based on different languages, namely Clojure and Haskell. The abstract nature of level graphs allows us to generate code in different languages.

3.4 Machine Learning for Benchmarking

In the previous two sections we have discussed multiple benchmarks in two different classes: hand-written benchmarks and randomly generated benchmarks. We have discussed the advantages and disadvantages of both. Hand-written benchmarks cost many person-hours to write and maintain, and are usually very limited due to IP. Random benchmarks can overcome the scarcity of hand-written ones at the cost of accuracy, since they are less realistic and, accordingly, not as useful for assessing how well a method will perform on real use-cases. There is a third approach that sits in-between the two above, which is to use machine learning to generate benchmarks with realistic properties. This section discusses this approach and its limitations.

3.4.1 *Generative models*

Machine learning models that could generate benchmarks fall under the general term "generative models". There are different classes of generative models, however:

1. A model in the **Fischer-Wald setting** is a machine learning model solving the problem of density estimation [Vap13]. This means finding a pdf $p'(t, \alpha_0)$ in a set of pdfs $\{p'(t, \alpha) \mid \alpha \in \Lambda\}$ parametrized by elements of the parameter set Λ, such that for the risk functional $R(\alpha) = \int -\log(p'(t, \alpha))dp(t)$, the value of $R(\alpha_0)$ is minimal over all $\alpha \in \Lambda$.
2. A **conditional estimation** model can again mean a solution to a few different problems in different settings [Vap13]. For a random

variable Y over code, it estimates either the joint distribution $X \times Y$ or one of the conditional probabilities $p(y \mid X = x)$ or $p(x \mid Y = y)$, where X is the random variable representing a piece of code (i.e. $X(\omega) = \omega$, the identity on Ω).

Conditional-estimation generative models have plenty of applications. For example, they can be used for code completion tasks, which could even be leveraged to create code that is close to a specified feature vector. With well-chosen features, this would allow for tools to create other kinds of benchmarks out of, e.g. a representative data set (see Section 3.3). Even more so, this could be used to create domain-specific benchmarks with more samples out of a small domain-specific dataset, by producing code with feature vectors as extracted from the small dataset. Tuning heuristics and even auto-completion tasks could all be based on conditional estimation models. The focus of this section is the discussion of using solutions to (1) for benchmark generation. This problem is the basis for generative models of code, and solutions to (2) are based on or related to it as well.

3.4.2 *Potential Problems*

Generative models learn to produce samples similar to those they have seen in the training data. They could then be leveraged to create arbitrarily large benchmark sets. The problem with this is that, in the ideal case for the Fischer-Wald setting, the code produced by the generative model should be indistinguishable from the training data. Concretely, it should be code that is also i.i.d. with respect to the (implicit) pdf of code. If this training set is available, then it can be used instead of the synthesized benchmarks. Figure 3.5 illustrates this further.

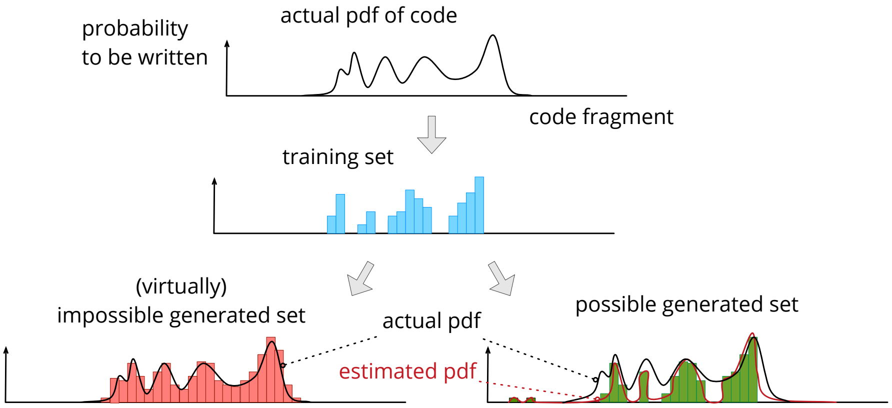


Figure 3.5: An illustration of generative models in the Fischer-Wald setting.

In the figure, the theoretical pdf depicted represents, again, the probability for a particular program to be written. Below it, the illustration represents a plausible histogram of the code actually present in a training set. Underneath it, two additional histograms are depicted. To the right, in green, a histogram that is plausibly synthesized by a good generative model trained with the training set. While it need not be identical to the training set, it should be similar to it, provided the generative model has been trained well. To the left, a histogram is depicted that illustrates a very implausible generated set. How should the generative model know of the unlikely cases it has not seen, and produce no synthetic code that would never be written by a human? By the definition in (1), the closer the histograms are to the depicted pdf, the better are the generative model and training set.

In case we want a "representative benchmark", it is thus not clear that a generative model like this is useful. The programs created by the generative model are i.i.d. with respect to some $p'(t) = p'(t, \alpha_0)$ that is different from p. For estimating the expected value $E[\mathcal{P}]$ of some pertinent property $\mathcal{P}$, we get additional accuracy by increasing the number of samples l we use to estimate it. This additional accuracy, however, could be canceled out by the error in p' (as quantified by, e.g. the risk functional $R(\alpha_0)$). Whether this is the case, of course, depends on the concrete problem and errors, and cannot be concluded generally. We believe, however, that it is probably the case for most instances of the state-of-the-art in generative models of code. It is likely that with the current state of generative models, where enough training data is present to train such a model, the raw data should be at least good enough, if not even better than synthetic data from the model.

For the other kinds of benchmark, the situation is less problematic. If we have a filter to distinguish the types of programs, e.g. one based on vectors of features interesting for the use case, then we can use generative models with these filters to produce benchmark sets of the kinds "representative coverage benchmark" and perhaps in some cases even "fuzzing benchmark", if the generative models generalize well and are run long enough to produce enough data.

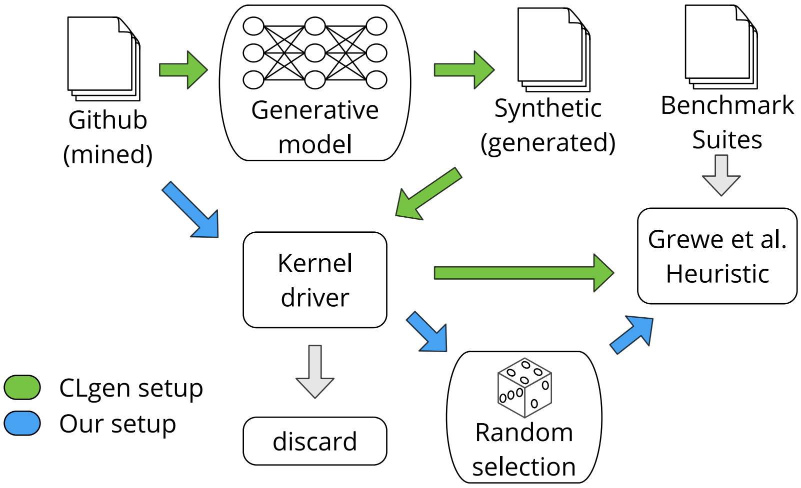


Figure 3.6: The flow of CLGen and our re-evaluation. Adapted from Figure 1 in [Goe+19].

We investigated these problems in [Goe+19], where we re-evaluated the benchmark generation of CLGen [Cum+17a]. Figure 3.6 summarizes the flow of CLGen and our re-evaluation. In the original CLGen setup, the authors of [Cum+17a] mine kernels from Github to train a generative model of code. The model in CLGen is a character-based model using a Long Short-Term Memory (LSTM) [HS97] architecture to learn a character distribution from (normalized) code. The generative model is then used to generate synthetic benchmarks. A driver ensures they compile and provides input values for the generated benchmarks, which are used together with a set of established benchmark suites to train a heuristic. The Grewe et al. heuristic [OWG13] trained from these benchmarks is a machine learning model that decides based on a set of hand-designed code features whether to execute an OpenCL kernel in a CPU or the GPU. Our setup takes the alternative route of foregoing the generative model, and using the mined github kernels instead of the synthetic benchmarks.

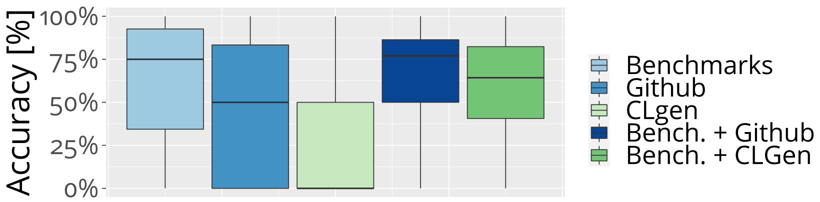


Figure 3.7: Accuracy obtained by the heuristic for the different datasets in the setup. Adapted from Figure 2 of [Goe+19].

We examined the code space and the pdfs from the different datasets. Figure 3.7 shows box plots comparing the accuracy obtained by the heuristic using the different datasets in both scenarios. In all cases, one of the benchmarks was excluded from the training dataset and used as test set. We repeated the experiment for all benchmark suites as test set. For the Github dataset we select a subset of the same size (1000) as the CLGen dataset used, which is part of the artifact of [Cum+17a]. We repeat a random selection of a subset of the Github data 100 times and report the median accuracy obtained this way, to control for the size of the dataset. We also consider both the enhanced datasets as in the original work [Cum+17a], namely the benchmarks enhanced with the CLGen kernels and alternatively enhanced by the original mined Github kernels. Finally we also consider using each dataset on its own for training, to help the comparison between the usefulness of the Github kernels and the generated CLGen kernels. It is obvious that the kernels generated by CLGen are not as useful for training as are the original Github kernels. In [Goe+19] we also empirically showed that adding more kernels did not help. We believe the problem is that the generated kernels are not *representative*. They are not as useful for maximizing $E[\mathcal{P}]$ as outlined above, $\mathcal{P}$ being the accuracy of the CPU/GPU mapping. We have to be careful with the conclusions we can draw from this. Particularly, upon closer examination (see [Goe+19]), the feature space of the Grewe et al [OWG13] heuristic seems to be rather ill-suited for the task.

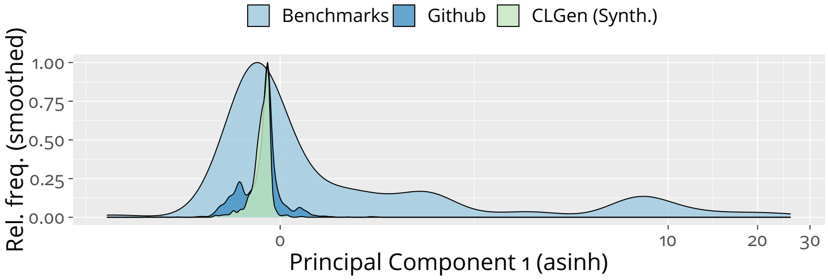


Figure 3.8: Smoothed relative frequencies of kernels as function of the first principal component. Adapted from Figure 6 of [Goe+19].

Figure 3.8 shows a smoothed estimation of the relative frequencies of kernels. The feature space is obviously not one-dimensional. We thus project it into one dimension for visualization by making a principal component analysis using all points. The figure shows the relative frequencies as a function of the first principal component, i.e. the one with the largest eigenvalue (by modulus). This figure thus serves to reproduce the intuition of representativeness as illustrated in Figure 3.5. It is very clear that the (feature) space covered by the benchmarks is larger than that covered by the Github kernels. These, in turn, cover more of the feature space than the generated CLGen kernels. These results are consistent with an explanation of the results from Figure 3.7, within the formalism as introduced here. Concretely, considering the formalism of benchmarks as reproducing a particular probability density and considering the task we want to learn as a random variable. We believe this probabilistic model of benchmarks has the potential to drive research forward in this direction, and we should focus on it in future work.

A clear first conclusion from this re-thinking of the benchmarking model is that we should also question the objective we are measuring in Figure 3.7. The accuracy we consider is the accuracy on the established benchmark suites. While this seems natural, the question is, is it the most useful objective? In a real-world scenario we will have our own codebase and will want to get the maximal accuracy in our code base. Good performance in the benchmarks is only useful to us if our code is similar to that on the benchmarks.

To evaluate this scenario, we took all 91 kernels from a concrete project, the Freedesktop project[4], and removed them form the Github dataset. The choice of the project is in principle arbitrary, the important property being that it has a moderate amount of kernels to evaluate on, without significantly reducing the Github dataset to the point we cannot use it. Using the same methods as above, we assessed the accuracy of training with all seven[5] benchmark suites compared to the Github kernels, without the Freedesktop kernels, obviously. Surprisingly, the heuristic performed significantly better with the Github kernels at 73%, compared to the 48% obtained with the established benchmarks. These results support the thesis that the concept of representativeness is central to benchmarking and models like the one proposed here should be investigated further.

3.4.3 *Models of Code*

One property of the generative models in CLGen is the way they represent code. They do so by considering the (normalized) code as a stream of characters that the model learns to predict. So far, in this thesis, we have strongly motivated graph-based representations of code, from dataflow graphs even to the closely-related level graphs. It is certainly not a new insight that graphs are well-suited to represent code in its non-linearity. Compiler construction in general is based on multiple graphs, like syntax trees or control- and data-flow graphs (CDFGs).

Based on this observation, we investigated graph-based representations of code for machine learning. We focused specifically on compilers [Bra+20]. Graph models in machine learning are an emerging field, with Gated Graph Sequence Neural Networks (GGNNs) [Li+15] being successful in multiple reasoning tasks. In the context of programming language models, GGNNs and related graph models have also been very

---

[4] https://www.freedesktop.org/

[5] the benchmark suites are: AMD SDK, NPB, NVIDIA SDK, Parboil, Polybench, Rodinia, SHOC

successful [ABK17; Cum+20; Ye+20; Pal+20]. These models use a iterative message-passing semantics to learn the dependencies from adjacent nodes in a graph. We designed code representations based on two graph representations of code common in compilers [Bra+20], CDFGs and abstract syntax trees (ASTs), as well as a general framework for investigating these kinds of code representations [BGC20]. Details about the machine learning architecture are beyond the scope of this thesis, here we just discuss the graph-based models and how they expose their semantics.

To test the different compiler-based graph representations of code, we trained [Bra+20] multiple deep learning models for the OpenCL CPU/GPU classification task described above. Figure 3.9 shows a comparison of the accuracy of the different graph-based models. The baseline model by Grewe et al. [OWG13] is the same baseline model from the previous CLGen experiments in this section. Additionally, we compare with two state-of-the-art deep learning models for this task, DeepTune [Cum+17b] and inst2vec [BJH18]. The graph-based models we evaluate are a control flow graph (CFG), and control- and data-flow graph (CDFG), enhanced with multiple edge annotations, for data dependencies of the return values of function calls (CALL) and store-load memory dependencies (MEM).

The graph shows two distinct evaluation setups. The first is the random split setup, which is the same one used by [Cum+17b; BJH18]. It joins all seven benchmark suites investigated into a large set of kernels and randomly splits it into 10 disjoint subsets. Each of the disjoint subsets is then used as the testing set, training with all other nine. We report the geometric means [FW86] of the accuracy.

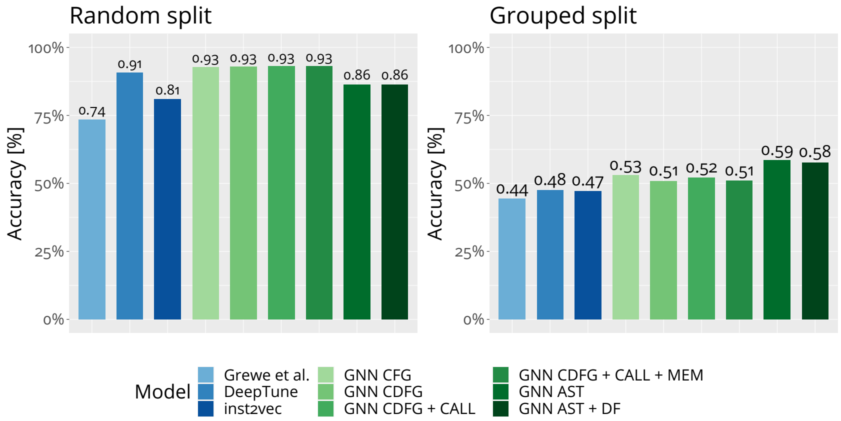


Figure 3.9: A comparison of the accuracy of multiple machine learning methods for the CPU/GPU classification of OpenCL kernels. Adapted from Figure 12 of [Bra+20].

A potential problem with the random split setup is that it mixes kernels from the same benchmark, which are likely to be more similar. This goes back to the representativeness argument for the Freedesktop evaluation above. A random split assumes a more representative benchmark for training. To investigate how this affects the algorithms, we change the split to also test how they generalize across benchmark suites. Instead of random disjoint subsets of equal sizes, we split the kernels for test/-training according to the benchmark suites. One suite is used for testing, all others for training, and we again report the geometric mean of the accuracy. This is the second setup, the grouped split.

The results in Figure 3.9 show that graph-based models achieved better accuracy than sequence-based ones, in general terms. This is not surprising, as it has been discussed that they are better at exposing the non-linear structure of code. Also not surprising is that all models do worse in the grouped split, when forced to generalize across benchmark suites. However, it is worth noting that the CDFG-based representations performed better on the random split, and the AST-based representations performed better on the grouped split. A CDFG is at a level of abstraction closer to the machine than an AST, which is closer to the code itself as written by a human. In this light, it is not surprising that a CDFG-based representation was better at learning with a more representative benchmark in the random split. The problem in that case is more related to the execution of code on a CPU or a GPU, whereas on the grouped split, for generalizing across benchmark suites, understanding the semantics of the code is more important for predicting how it might fare.

In this section we have seen how graph-based representations and the level of abstraction are important, as well as how we should pay closer attention to the representativeness of a benchmark. A natural question at this stage is whether this insight can be used to improve generative models and generate better, more representative benchmarks. For this we also need graph-based generative models, which have received less attention than GGNNs in inference. The graph generative model of [Li+18b] works by generating sequences that construct the graph. While this allows us to create graphs representing code, the sequential structure of the generative sequences still pose some problems. This model is also very generic, which makes it easy to generate invalid code graphs, just like CLGen can generate invalid code. Expanding upon this, Alexander Brauckmann managed to generate more valid code samples than CLGen [Bra20] (up to 88%, compared to 38% for CLGen). In a related effort, Alexander Thierfelder designed a domain-specific extension to the model of [Li+18b], aiming to generate LLVM-based graphs that are correct by construction [Thi20]. The LLVM language is complex, and we could unfortunately not design a generative model where the graphs are correct by construction, but we could capture most of the LLVM semantics in the model. Graph-based models of code are a promising direction for future work, which could allow us to generate representative benchmarks, among others [LC20].

4

MATHEMATICAL STRUCTURES IN MAPPINGS

The space of mappings in the software synthesis flow we described has a rich mathematical structure. This chapter aims to explore and expose that structure, at least in part. We will consider two main aspects of the mathematical structure hidden within the simple notion of a mapping, namely the inherent symmetry, and the degrees of similarity between mappings. We will consider how to extract this structure in a computationally-efficient fashion, and how it can be exposed to tools that aim to exploit it, in different representations.

This thesis focuses primarily on a view of the mapping problem centered on computation, instead of data. In many cases, with the increasing discrepancy between execution frequency and memory access times (cf. Figure 1.1) this view is not ideal. The problem space of data allocation is usually more clearly structured and can be modeled better. For example, we worked on integer linear programming (ILP)-based methods to describe and optimize memory allocation [Ode+14; Ode+15; GCL16]. We omit this work from this thesis for space reasons. We also omit work on emerging memory technologies, concretely race-track memory (RTM), where we used similar ILP-based models and other meta-heuristics like genetic algorithms or domain-specific heuristics to optimize data placement [Kha+20].

4.1 Symmetries

In this chapter we will explore the mathematical structure of symmetry in the software synthesis process, mostly the work published in [GC15; GSC17; Goe+17; GMC18; GNC]. The material in this section makes use of concepts in group theory. We assume the basic concepts as seen in any undergraduate course on group theory, with the definitions of groups, actions and orbits. A brief introduction, to the level required by this chapter, can be found in Appendix A.1.

4.1.1 *Architectures and Applications*

Intuitively, when we say an object is very symmetric we usually mean it has parts that are similar or identic, and the object looks identical (or similar) from multiple points of view. In a symmetric face, for example, both the left and right sides of the face are similar. A hexagonal mosaic might look the same when seen from six different angles. Mathematically, this is commonly modeled through transformations. A reflection along the vertical axis in a face, or rotations of 60° in the heaxgon, both leave the object (mostly) unchanged. We can do the same for hardware architectures, even heterogeneous ones.

For example, the Exynos 5 in the Odroid-XU4 has four identical Cortex A7™, say $\mathrm{PE}_1, \dots, \mathrm{PE}_4$ and four identical Cortex A15™cores, say $\mathrm{PE}_5, \dots, \mathrm{PE}_8$. A transformation that swaps the cores PE_1 and PE_2 leaves the archtiecture topology unchanged, since the cores are identical. This is depicted in Figure 4.1. On the other hand, a transformation that swaps

PE_1 and PE_5 does change the topology, since the cores are of different types. As can also be seen in Figure 4.1.

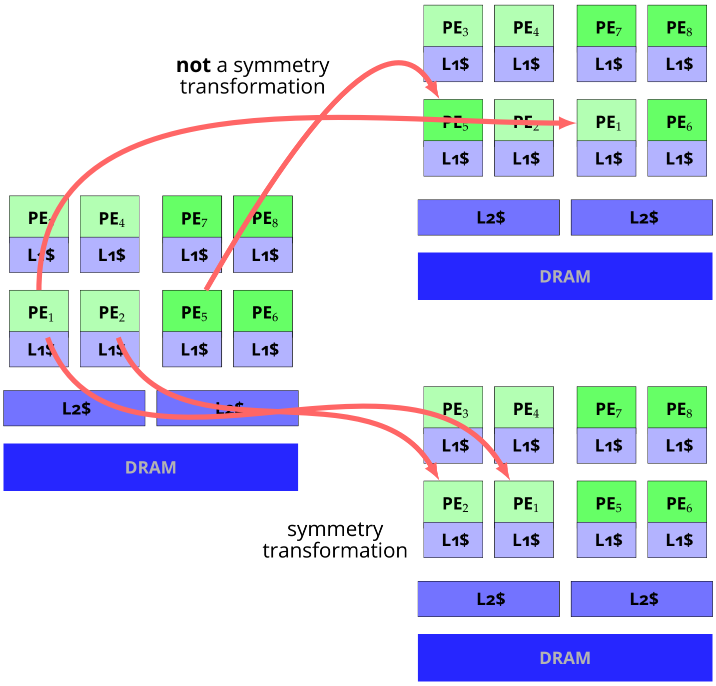


Figure 4.1: Examples of transformations in the Odroid-XU4 architecture.

When the interconnect subsystem is more complex, this is also reflected in the topology. Consider the NoC-based architecture depicted in the example, with four identical cores $\mathrm{PE}_1 \ldots, \mathrm{PE}_4$. An analogous transformation to the one described before, which swaps the cores PE_1 and PE_2, is not a symmetry of this topology, as depicted in Figure 4.2. The change in the cores changes the communication patterns. Before the transformation, sending data from PE_1 to PE_3 needs two hops, whereas after the transformation it can be sent within a single hop, as shown by the red paths in the NoC.

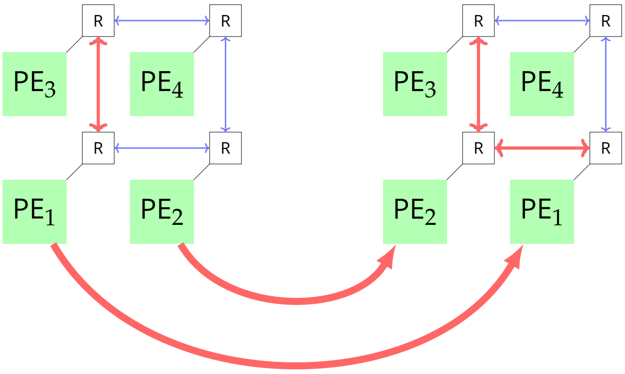


Figure 4.2: The communication topology affects symmetries in architectures.

Generally, the transformations that preserve the structure of the architecture topology have a clear structure. If two transformations t_1 and t_2 preserve the structure of the architecture topology, then their composition $t_1 \circ t_2$ also preserves it. Similarly, it is clear that reversing a transformation t_1^{-1} also preserves the structure. Finally, the identity transformation on the architecture id_A (which does not change anything) clearly preserves the structure. These observations together mean that these transformations have the structure of a group with the function composition $(\circ)$ as its operation.

More precisely, the group of symmetries of the architecture is precisely the group of graph isomorphisms from the architecture graph A to itself. An isomorphism from an object to itself is called an automorphism. We denote the group of automorphisms of the architecture A as $\mathrm{Aut}(A)$

For the case of the NoC-based architecture, the authomorphism group $\mathrm{Aut}(A_{\mathrm{NoC}}) \cong D_4$ is a dihederal group on 4 points. It consists of 3 rotations, 4 reflections and the identity transformation. The Odroid architecture, on the other hand, has $\mathrm{Aut}(A_{\mathrm{Odroid}}) \cong S_4 \times S_4$ as symmetry group. This group with 48 transformations consists of (independent) arbitrary permutations of the A15 and A7 cores.

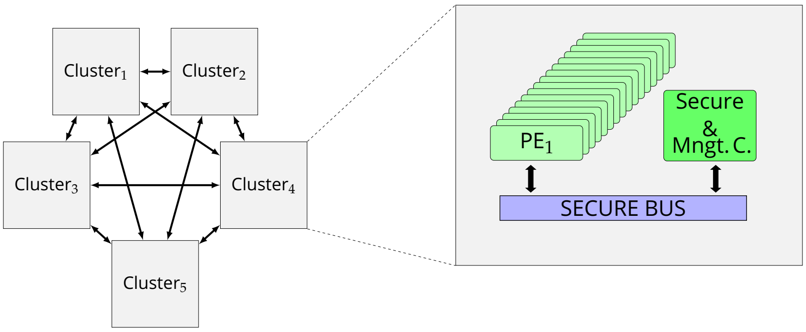


Figure 4.3: The topology of the Kalray MPPA3 Coolidge.

Since the Odroid architecture is heterogeneous, both clusters are distinct and there is no symmetry between them. Many complex architectures, however, do consist of multiple identical clusters. Consider the architecture depicted in Figure 4.3. It is the MPPA3 Coolidge from Kalray [inc20] and consists of 5 identical clusters. Each cluster has 17 cores, 16 of which are identical general-purpose cores, and the last one is a special-purpose secure and management core.

The MPPA3 Coolidge is a hierarchically-designed architecture. The five identical clusters are conceptually at a different level than the cores at each cluster. Designs like the HAEC [Fet+19] topology mentioned in the introduction (cf. Figure 1.2) have even more levels of hierarchy. The symmetries of these hierarchical architectures are reflected in the different levels of hierarchy of the topology [GNC]. For example, the automorphism group of the MPPA3 Coolidge is $\mathrm{Aut}(A_{\mathrm{Coolidge}}) \cong S_{16} \wr S_5$ and has $16! \cdot 5! \approx 2.51 \cdot 10^{15}$ symmetries.

So far we have discussed the symmetries of architectures. However, we can apply the same principle to applications and their graphs. Conisder the audio filter example application from Section 2.1 (cf. Figure 2.1). The left and right channels perform precisely the same computation on different data. We could not, for example, just swap the `fft_l` and `fft_r` nodes, since that would result in a different application that also swaps the channels of the audio file. On the other hand, if we swap the whole subgraph consisting of `fft_l`, `filter_l` and `ifft_l` with the equivalent subgraph of `fft_r`, `filter_r` and `ifft_r`, the application remains identical. This is depicted on Figure 4.4.

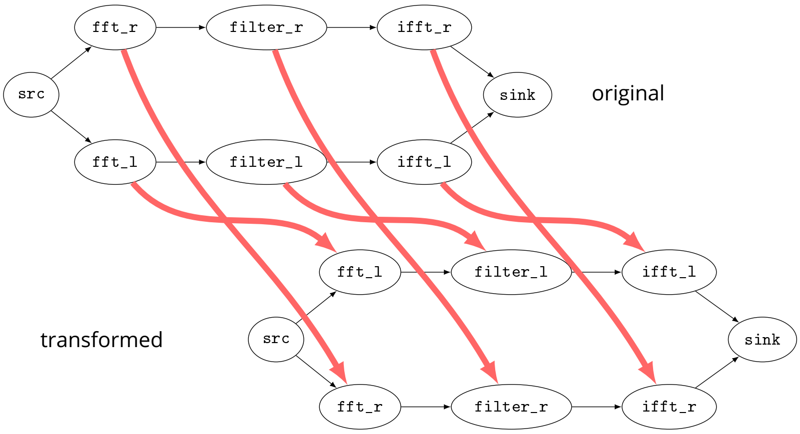


Figure 4.4: A symmetry transformation of the audio filter application.

Mathematically, we need to model the semantics of the application to reflect its symmetries. For an application $K = (V_K, E_K)$ we can label the nodes V_K with unique identifiers relating them to the KPN process that execute them (e.g. the `__PNprocess` in a CPN program). Formally, thus, the automorphism group $\mathrm{Aut}(E)$ of the labeled graph K is trivial, i.e. $\mathrm{Aut}(E) = \{\mathrm{id}\}$. We could label K differently to capture the symmetry from Figure 4.4. For example, if we use the source code of the process as label, we would capture this symmetry. We have to be careful, however, as this can lead to a problematic definition of symmetries.

An application might use the same code at different points, resulting in very different behavior. For example, consider an application that receives a list of points, which it sorts before operating on it. Before returning the list, it sorts them again to ensure they are sorted. Both times it sorts the list using the quicksort algorithm, yet the second time the list is almost always sorted or close to being sorted. Then, the execution of the same quicksort code in the second instance behaves very differently from the first time.

A difference like the one outlined above is very difficult to capture automatically, as it requires understanding of the application to a very high level of abstraction. We thus consider application symmetries as manually-defined annotations. There are some conceivable ways to automatically capture and annotate such application symmetries, for example when dealing with known data-level parallelism (DLP). In future work, a framework as we discuss in Chapter 6, Section 6.2 could be extended to extract application symmetries from DLP. For the rest of this thesis, however, we focus on symmetries induced from the architecture.

4.1.2 *Mappings*

We have seen how the architecture and applications have symmetries in their structure. The groups $\mathrm{Aut}(A)$ and $\mathrm{Aut}(K)$ act on the architecture A and the application K, respectively. These actions also induce an action on the mapping space. Let $m : K \to A$ be a mapping. Recall that a symmetry $\sigma \in \mathrm{Aut}(A)$ of the architecture is a transformation that leaves the structure of the architecture unchanged. Conisder then the mapping

$\sigma m := k \mapsto \sigma(m(k))$. Since the structure of A is unchanged, then the structure of m and σm is also identical. All observable properties Θ of m and σm, like the execution time or energy consumption, are the same. If they were not, it would have to be due to a structural difference in the (sub)architectures $m(K), (\sigma m)(k) = \sigma(m(K)) \leqslant A$, which are isomorphic by assumption on σ. We say that these properties like the execution time and energy consumption are *invariants* of the group action.

The case for K is analogous. Let $\pi \in \mathrm{Aut}(K)$ be a symmetry of the application. Then the mapping $\pi m := k \mapsto m(\pi^{-1}k)$ is equivalent to m. Note that we define it with π^{-1} instead of π so that this defines a left action. Indeed, for $\pi, \tau \in \mathrm{Aut}(K)$, we have

$$(\pi(\tau m))(k) = \pi m(\tau^{-1}k) = m(\pi^{-1}\tau^{-1}k) = m((\tau\pi)^{-1}k) = ((\tau\pi)m))(k)$$

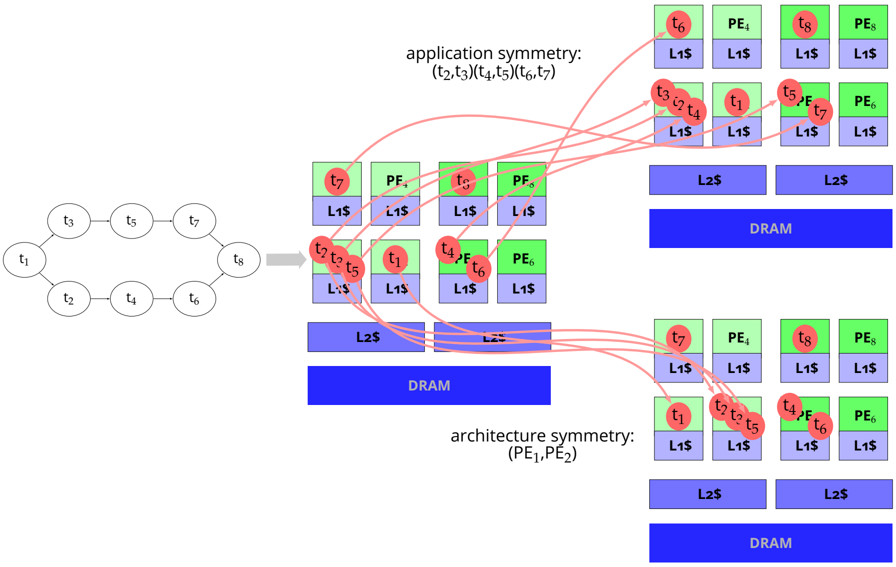


Figure 4.5: Group actions on mappings.

Figure 4.5 shows an example of the action on mappings. It depicts a mapping $m = [\mathrm{PE}_2, \mathrm{PE}_1, \mathrm{PE}_1, \mathrm{PE}_5, \mathrm{PE}_1, \mathrm{PE}_5, \mathrm{PE}_3, \mathrm{PE}_7]$ of the audio filter application on the Odroid XU4. On the bottom right, we depict the action of σ on m for the architecture symmetry $\sigma = (\mathrm{PE}_1, \mathrm{PE}_2)$, in cycle notation[1], which is the same symmetry transformation depicted in Figure 4.1. This results in the mapping $\sigma m = [\mathrm{PE}_1, \mathrm{PE}_2, \mathrm{PE}_2, \mathrm{PE}_5, \mathrm{PE}_2, \mathrm{PE}_5, \mathrm{PE}_3, \mathrm{PE}_7]$. On the top right, we show the action of π on m for the application symmetry

$$\pi = (\texttt{fft\_l}, \texttt{fft\_r})(\texttt{filter\_l}, \texttt{filter\_r})(\texttt{ifft\_l}, \texttt{ifft\_r}),$$

which is the application symmetry depicted in Figure 4.4. This results in the mapping $\pi m = [\mathrm{PE}_2, \mathrm{PE}_1, \mathrm{PE}_1, \mathrm{PE}_1, \mathrm{PE}_5, \mathrm{PE}_3, \mathrm{PE}_5, \mathrm{PE}_7]$.

From the underlying problem formulation, as defined in Chapter 2, mappings under these symmetries are necessarily indistinguishable from each other, since they rely on inherent symmetries of the models. On the other hand, these are just models, they need not reflect reality. It still leaves the question open, how does this hold up in pracitice? Are equivalent mappings actually equivalent? In [Goe+17] we tested this empirically, by executing four equivalent mappings and measuring the runtime and

---

[1] See Appendix A.1 for an explanation

energy. We tested four equivalent mappings of the audio filter benchmark on an Odroid XU3 and executed each 50 times, measuring three metrics. The Odroid XU3 is almost identical to the Odroid XU4, but features on-board energy sensors. We measured the CPU time, which is the total aggregate time spent by all CPUs executing the application. We also measured the wall-clock time, which is the total time of the application, measured from start to finish as it would pass on a wall clock. Finaly, we measured the total energy over the INA-231 sensors connected over the I2C bus, as the aggregate of the energy measured at each individual component, sampled at 10 Hz.

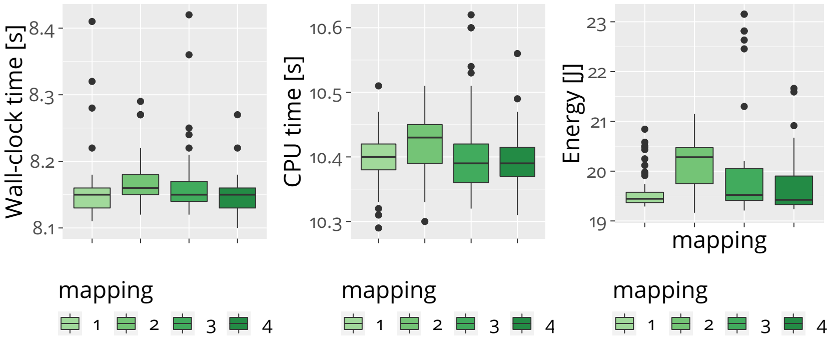


Figure 4.6: Measurements of four equivalent mapings for the audio filter application on the Odroid-XU3 architecture.

Figure 4.6 shows the results of the measurements as box-and-whiskers plots. We see that the execution times of all mappings are well within the ranges of each other, both when measured as wall-clock or CPU time. We can test this in a more rigurous fashion with an Analysis of Variance (ANOVA) F-test. For the wall-clock data, we get a probability of $p = 5.06\%$, rejecting the null hypothesis that variance in the wall-clock times from different mappings is explained by more than statistical variance. In other words, the variance can be considered to be just statistical noise. For the CPU time data, the value of $p = 15.2\%$ it is even clearer that the variance is likely just statistical. We can conclude that, at least for this example, equivalent mappings indeed seem to have the same execution time behavior.

The energy consumption of the mappings is less clearly distinct, especially for the first two mappings, yet they all still require comparable amounts of erengy. An ANOVA F-test yields a p value of 0.009%. It is worth noting here that these measurements are just a first estimation. The accuracy of the energy data depends on a multitude of factors like the sampling rate or the measurement accuracy of the INA-231 sensors. We have not made a proper assesment of measurement errors and error propagation, which is especially large in the energy measurements that span measurements from four different components. Precise energy measurments, however, are beyond the scope of this thesis. With this experiment we cannot really accept nor discard the hypothesis that energy consumption is an invariant of the mapping symmetries.

4.1.3 *Calculating Symmetries*

The branch of mathematics known as computational group theory deals with computational aspects of the theory of groups, which we have used here for formalizing the symmetries of applications, architectures and mappings. An overview of the methods of computational group theory can be found in [Hol05; Ser03], which both cover far beyond the basics presented in this subsection. Here we will present only the methods necessary for the calculations required of applications to software synthesis.

To leverage the symmetries explained in this thesis, we need methods to calculate the following:

1. Given an architecture graph A, calculate (generators for) the group of symmetries $\mathrm{Aut}(A)$.
2. Given a mapping $m : T \to A$ and the symmetry group $G := \mathrm{Aut}(T \to A)$, enumerate the orbit Gm.
3. Given two mappings $m, m' : T \to A$ and the symmetry group $G := \mathrm{Aut}(T \to A)$, determine whether $m = gm'$ for a $g \in G$, i.e. if the two mappings are in the same orbit.

Mature software exists for computational group theory that can, in principle, solve these problems. The GAP system is a Domain-Specific Language (DSL) for computational discrete algebra with a focus on (computational) group theory [GAP20]. We developed algorithms for dealing with problems 1-3 in GAP [GSC17]. We also included naive versions of most algorithms implemented directly in Python in `mocasin`.

Using GAP-based algorithms in software synthesis tools like MAPS in practice, however, comes with a series of complications. The largest problem is that it adds a dependency on a whole ecosystem. A complete distribution of GAP is over 200 MiB of size and takes around a second to start up in standard commodity hardware of today. Additionally, to communicate with a running GAP instance we need to use OS pipes, which is cumbersome and not portable. We thus developed a standalone library [GNC; Nic20], `mpsym`, which implements the required algorithms to solve problems 1-3 and includes a domain-specific extension for efficiently dealing with hierarchical (e.g. clustered) archictures [GNC].

Calculating the group of symmetries from an architecture graph (Problem 1) is very related to the graph isomorphism problem. This is a problem in NP, and it is not known, neither believed to be in P nor NP-complete. In December 2015, Lásló Babai published a pre-print where he claims to have found a quasi-polynomial algorithm [Bab16], yet at the time of this writing (January 2021) the peer-review is still not complete. Regardless of the worst-case complexity, graph isomorphism can be solved efficiently in practice for most instances [MP14]. Algorithms for doing so are implemented in nauty/Traces, which `mocasin` and `mpsym` use to solve Problem 1.

Virtually all MPSoCs have topologies that follow a well-defined set of design principles, like using NoCs, hierarchical clusters or groups of identical PEs. This is also the idea behind the `platform designer` module in `mocasin`. In [GNC] we showed that we can leverage this to construct the automorphism group of the architecture. In particular, the automorphism groups of hierarchical architectures are the wreath product of symmetries of the clusters. We used a specialized algorithm that leverages a wreath-product decomposition, originally applied in model checking [DM09]. Table 4.1 shows the domain-specific approach to finding architecture symmetries in hierarchical designs, as described in [GNC].

Most algorithms in computational group theory use a special data structure describing the group. This data structure is called a *base and strong*

Table 4.1: Correspondence of architecture and group-theoretic constructions. Adapted from Table 1 in [GNC].

Hardware Architecture	Group Theory
Bus-based connection (n identical elements/clusters)	Symmetric Group S_n
Distinct elements/clusters	Direct product $G_1 \times \ldots \times G_n$
NoC Connection with topology graph Γ (identical elements)	Automorphism group of Γ $\mathrm{Aut}(\Gamma)$
Hierarchical composition	Wreath product $G \wr H$

generating set (BSGS), see [Hol05; Ser03] for more details. The standard algorithm for calculating the BSGS for a group is the Schreier-Sims Algorithm. Multiple variants of this algorithm exist, which are more efficient under different circumstances. Computer algebra systems (CAS) like GAP use different variants with sophisticated heuristics for selecting which variant to use. In `mpsym` we implement some variants of the Schreier-Sims algorithm with a less sophisticated selection heuristic, which do not surpass GAP's performance. For all groups investigated in this thesis, however, `mpsym` was comparable to GAP, without the large ecosystem dependency [Nic20; GNC].

Problem 2 is a standard problem in computational group theory. We solve it using the Orbit algorithm, which can easily be adapted to a lazy variant, described in Algorithm 1. If we use a perfect hash, the algorithm returns exactly the orbit of the mapping. If the hash can have duplicates, a smaller orbit might be returned, but the algorithm will clearly never yield elements from outside the orbit. This lazy variant is especially useful when looking for any mapping in the orbit which fullfils some properties, instead of being interested in the full orbit. This is especially useful in the TETRiS system, which we will describe in Section 5.5.

Algorithm 1 A lazy variant of the standard orbit algorithm

```
input: A generating set $X = (g_1, \ldots, g_n), \langle g_1, \ldots, g_n \rangle = \mathrm{Aut}(M)$ for the mapping space, a mapping $m_0$.
output: The orbit of $m_0$: $\mathrm{Aut}(M)m = \{gm_0 \mid g \in \mathrm{Aut}(M)\}$
H ← $\{\mathrm{Hash}(m_0)\}$
CurElems ← $\{g_i m_0,\ \mathrm{Hash}(g_i m_0) \notin H \mid i = 1, \ldots, n\}$
H ← $H \cup \{\mathrm{Hash}(m) \mid m \in$ CurElems$\}$
while CurElems $\neq \varnothing$ do
    for $m \in$ CurElems do
        yield $m$
    CurElems ← $\{g_i m,\ \mathrm{Hash}(g_i m) \notin H \mid m \in$ CurElems$, i = 1, \ldots, n\}$
    H ← $H \cup \{\mathrm{Hash}(m) \mid m \in$ CurElems$\}$
```

Finally, to solve Problem 3, we could simply solve Problem 2 for both elements and see if the orbits are identical. Orbits form a partition of the mapping space M, meaning that two orbits are either identical or dijsoint (and the union of all orbits yields M). This is a very inefficient way of solving Problem 3, since it means we have to enumerate the whole orbit for each element. Using the same principle of the Orbit's partition, we can also just enumerate the orbit for one element and see if the other element is in it. While this is also an improvement, it is still very inefficient. Orbits can be very large when the problem has much symmetry. By default, `mpsym` uses this variant as a fall-back method to solve Problem 3 when correctness needs to be guaranteed.

Another alternative for this which works without enumerating any orbits is based on the fact that the symmetries of the mapping $\mathrm{Aut}(M) \leqslant \mathrm{Sym}(M)$, the symmetric group on M (i.e. the group of all permutations on M). Thus, if two mappings m, m' are in the same orbit under $\mathrm{Aut}(M)$, then they are also in the same orbit under $\mathrm{Sym}(M)$: there exists a permutation $\sigma \in S_{|M|}$ which takes m to m'. However, $|M|$ as we have seen can be very large, as it grows (at least) as $|V_A|^{|V_K|}$ and $\mathrm{Sym}(M) \cong S_{|M|}$ is thus unimaginably large, namely $|\mathrm{Sym}(M)| = |M|! \geqslant (|V_A|^{|V_K|})!$. If we consider only architecture symmetries, this all works over the much smaller $\mathrm{Aut}(A)$. We obviously do not have to iterate over the group to construct σ, since we know both m, m' we can construct it directly. Knowing σ, we can efficiently solve the *group membership problem* [Ser03] for these permutation groups, using the BSGS data structure. We know, namely, that σ is in $\mathrm{Aut}(M)$ if and only if $\mathrm{Aut}(M)m = \mathrm{Aut}(M)m'$, by the definitions of the orbit and σ. On the other hand, if we cannot construct σ from m, m', because it leads to contradictions, then, obviously, the orbits are different.

There is an alternative variant of this which also allows us to select a mapping to work with, e.g. for DSE. It is based on canonical representatives [GSC17; GMC18]. A canonical representative of an orbit Gm is an element $m_0 \in Gm$ such that there is a function $f : M/G \to M$ which maps Gm to m_0. In other words, the function f selects a unique element of every orbit, this element is the canonical representative.

For constructing our canonical representatives, we order mappings using the lexicographical ordering. For two mappings $m = (m_1, m_2, \ldots, m_k)$ and $m' = (m'_1, \ldots, m'_k)$ we say that $m \leqslant m'$ if and only if there exists a $j \leqslant k$ such that $m_i = m'_i$ for all $i < j$ and $m_j \leqslant m'_j$. The function f for the canonical element of the orbit thus maps Gm to $\min Gm$. In other words, we choose canonical elements to be the lexicographical-minimal elements of the orbits.

Algorithm 2 Local search for finding canonical representatives. Adapted from Algorithm 1 of [GMC18].

```
input:  A mapping $m$, a generating set $S$, with $\langle S \rangle = \mathrm{Aut}(M)$.
output: A mapping $m_{\text{canonical}} = gm$ with $m_{\text{canonical}} \leqslant m'$ for all $m' \in Gm$
$F \leftarrow \{m\}$
$F_{\text{old}} \leftarrow \varnothing$
while $F \neq F_{\text{old}}$ do
    $F_{\text{old}} \leftarrow F$
    for all $s \in S$ do
        for all $m' \in F$ do
            if $sm < m$ then
                $F \leftarrow F \cup \{sm\}$
    $F \leftarrow \{\min_{m' \in F} m'\}$ (optional)
return $\min_{m' \in F} m'$
```

To find the lex-minimal canonical representatives we use a local-search algorithm based on an iteration similar to the Orbit Algorithm. Algorithm 2 shows this local-search heuristic. This algorithm returns the lex-minimal element of the orbit if the generating set has a particular property, which we called being a *strictly order-preserving generating set* [GMC18]. We say S is a strictly order-preserving generating set if for two mappings $m' < m$ in the same orbit, i.e. $m \in \langle S \rangle m'$, there exists a word $s_1, \dots, s_n$ in the generators $s_i \in S$, such that $m' = s_1 \dots s_n m$ with $s_i(s_{i+1} \dots s_n)m < (s_{i+1} \dots s_n)m$ for all $i = 1, \dots, n-1$. For example, for the symmetric group S_n, the set of all transpositions $S = \{(i,j) \mid i \neq j \in \{1, \dots, n\}\}$ is such a strictly order-preserving generating set. Without this property, the local search could yield an element which is not the lex-minimal element. If we remove the optional reduction in Line 9, we significantly speed up this search and make the probability of finding only a local minimum instead of the global one higher. Since all mappings in the orbit are equvialent, such a local minimum will always have the same objective properties Θ as the real canonical representative (cf. Section 2.4). Thus, finding a local minimum instead of the canonical representative is tantamount to considering a smaller group of symmetries, and thus a very acceptable risk for a considerable speed-up. Both `mpsym` and `mocasin` implement this heuristic and use it by default for design-space exploration, as we will see in Section 5.3. We also integrated `mpsym` into `mocasin`, using the simple Python versions of the algorithms in `mocasin` only as fall-back.

4.1.4 *Partial Symmetries*

The symmetries we have considered so far can be considered as “global” symmetries: they are transformations of the complete structure (e.g. architecture, mapping). The intuitive notion of symmetry, however, is more general than this. What we consider as symmetry also includes the relationship of a structure to its parts. In particular, a symmetry can be local to a part of the structure, without being global. A general discussion of this can be found in [Law98], as well as a detailed exposition of the mathematical background of this section.

We can see what we mean by local structures in the example depicted in Figure 4.7. It shows two NoC architectures both with a regular mesh topology. The first one is a two-by-two mesh, the second one four-by-four. We can compare now the symmetries of both architectures intuitively, and see how these translate to the group-theoretic sense. The four-by-four mesh is larger, and has a sort of self-similarity: it can be thought of as composed of four copies of the two-by-two mesh arranged in a larger two-by-two mesh. Intuitively, thus, this four-by-four mesh has more symmetry.

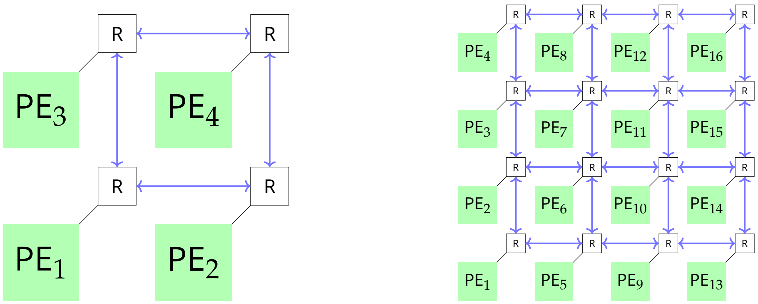


Figure 4.7: A comparison of the two different-sized meshes and the intuitive notion of their symmetries.

However, if we look at the group of automorphisms of the corresponding architecture graphs, we get a result that defies this intuition: both architectures have the same groups of symmetries! More precisely, their groups of automorphisms are isomorphic, they are dihedral groups on 4 points, D_4. More concretely, there are only 8 possible structure-preserving transformations acting on these two topologies, which are the rotations of $90°, 180°, 270°, 360° = 0°$ and the reflections among each of the axes (horizontal, vertical and both diagonals). We cannot, for example, divide the four-by-four mesh into a two-by-two mesh of two-by-two meshes, and rotate that larger two-by-two mesh by $90°$ or one of the smaller ones by $90°$. These two operations both work locally, if we ignore the rest of the structure, but do not preserve the whole structure of the mesh, as illustrated by Figure 4.8. The figure shows how a rotation on the bottom left 2×2 mesh breaks the communication structure. Highlighted is the communication between PE_1 and PE_3, which changes from 2 hops to 1 hop in the transformation.

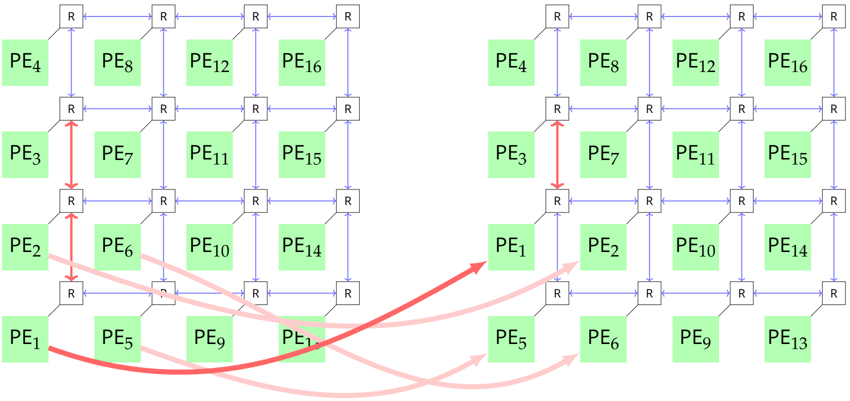


Figure 4.8: An example of a local symmetry that is not a global symmetry of a 4×4 mesh.

For the mathematical formalization of this intuitive notion of local symmetries, in this section, we follow [Law98]. There are essentially two equivalent ways of formalizing this intuitive notion of local symmetries, inverse semigroups and ordered groupoids. We will consider the formalization using inverse semigroups, as it is conceptually simpler for computations, and mathematically equally as powerful. In the case of global symmetries, there are concrete transformations of architectures and mappings, which correspond to abstract groups. For partial symmetries, we will consider partial transformations of mappings, which we will model as partial permutations, and these partial permutations (transformations) have a corresponding abstract inverse semigroup.

We start by defining partial functions and partial permutations.

Definition 4.1.1. Let X,Y be sets. A *partial function* $f : X \to Y$ is a function from a subset of X to a subset of Y. We denote the domain of f by $\mathrm{dom}(f)$ the codomain of f by $\mathrm{cod}(f)$. Thus, the partial function $f : X \to Y$ is a (total) function $f : \mathrm{dom}(f) \to \mathrm{cod}(f)$

Definition 4.1.2. Let X be a set. A partial function $f : X \to X$ from X to itself is called a *partial permutation* if the (total) function $f : \mathrm{dom}(f) \to \mathrm{cod}(f)$ is a bijection between $\mathrm{dom}(f)$ and $\mathrm{cod}(f)$.

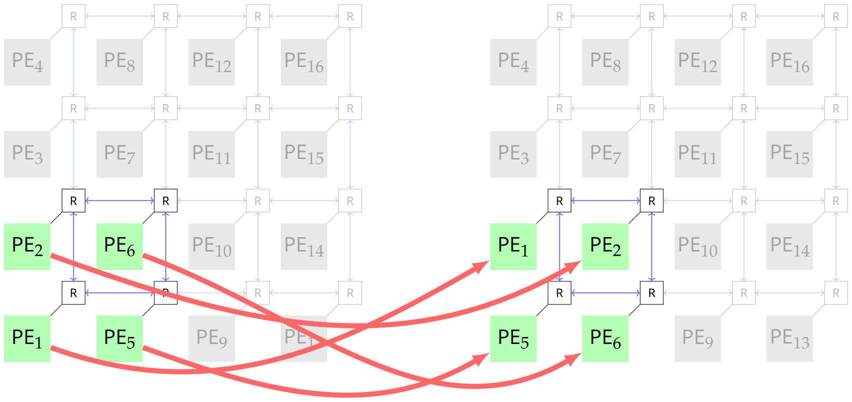


Figure 4.9: The transformation of Figure 4.8 as a partial permutation.

We can think of partial functions thus as functions that are not defined everywhere, and partial permutations, accordingly, are not defined everywhere. For example, the partial permutation $f : \{1, \ldots, 16\} \to \{1, \ldots, 16\}$ defined as $f(1) = 2, f(2) = 6, f(5) = 1, f(6) = 5$ is a rotation of the bottom-left 2×2-mesh in the 4×4-mesh depicted in Figure 4.8, but is not defined on the rest of the architcture. This is a partial symmetry of the 4×4-mesh. This partial permutation is depicted in Figure 4.9. We can write it also as:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 2 & 6 & - & - & 1 & 5 & - & - & - & - & - & - & - & - & - & - \end{pmatrix}$$

Because the set $\{1, \ldots, 16\}$ is understood from context, we can also write it, shorter, as:

$$\begin{pmatrix} 1 & 2 & 5 & 6 \\ 2 & 6 & 1 & 5 \end{pmatrix}$$

We also use a notation similar to the cycle notation of group theory, where we use a cicle with round brackets to denote a full cycle, where the last element maps to the first. Square brackets to denote when this is not the case, i.e. the function is not defined on the last element of that cycle. In this notation, singleton cycles cannot be omitted as in the case of groups. In other words, fixed points have to be represented as one-element cycles. The partial permutation from Figure 4.9 can thus be written much more compactly as: $(1, 2, 6, 5)$. This is a full cycle, but it is only defined on the subset $\{1, 2, 5, 6\}$. In the group context, the cycle $(1, 2, 6, 5)$ as an element of the symmetric group on 16 points, would instead mean the (complete) permutation that fixes $\{3, 4, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$. As a partial permutation in cycle notation we would write this as:

$$(1, 2, 6, 5)(3)(4)(7)(8)(9)(10)(11)(12)(13)(14)(15)(16)$$

A different example of a partial permutation is the partial permutation that moves the first column to the right (and is not defined on the rest), as a cycle: $[1, 5][2, 6][3, 7][4, 8]$. On the other hand, the partial permutation that is a diagonal reflection on the bottom-left 2×2 (sub)mesh is, in cycle notation, $(2, 5)(1)(6)$. These two partial permutations can also be written in the matrix notation from above as:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 5 & 6 \\ 1 & 5 & 2 & 6 \end{pmatrix}$$

For computations [Eas+19], the three notations can be interpreted to make different data structures that make different opperations more efficient, like application of the partial function (as an array look-up), for sparse partial permutations (as lookups in key-value pairs), or cycles for efficient multiplication (as concatenation). They have different benefits and drawbacks. For readability though, the cycle notation is the most compact one, and the one we will use for the rest of this thesis.

Just as for groups, we can define the (left) action of a semigroup:

Definition 4.1.3. Let S be a semigroup and X be a set. We say that S *acts* on X (on the left) if there is a function $\cdot : S \times X \to X$ such that $(ab) \cdot x = a \cdot (b \cdot x)$. If S is a monoid with identity 1 and the function $\cdot$ satisfies the condition $1 \cdot x = x$ for all $x \in X$, we say that the action is a monoid action.

The action of a semigroup of partial permutations on an architecture works the same as with groups, except it does not work on the whole architecture. Let f be a partial permutation on an architecture A, and $m : K \to A$ be a mapping on that architecture. If the partial permutation is defined on all cores that m maps to, i.e., $\mathrm{im}(m) \subseteq \mathrm{dom}(f)$, then we can use the action of the semigroup of partial permutations of A to define another mapping fm by $fm(t) = f \cdot m(t)$ for all t in K. If f is not defined on some of the cores of m, i.e., $\mathrm{im}(m) \nsubseteq \mathrm{dom}(f)$, then we cannot define fm. In this way, f also defines a partial permutation $\hat{f}$ on the set of mappings $M \subseteq \{m : K \to A\} =: A^K$.

Consider for example the mapping of an application with three tasks to the 4×4-mesh defined by $m_1(t_1) = m_1(t_3) = \mathrm{PE}_1$ and $m_1(t_2) = \mathrm{PE}_5$, which we can also write as the vector $m_1 = (1, 5, 1)$. Then the partial permutation $(1, 2, 6, 5)$ from Figure 4.9 above defines the mapping $(1, 2, 6, 5)m_1 = (2, 1, 2)$. Similarly, the action of the partial permutation $(2, 5)(1)(6)$ yields a new mapping, $(2, 5)(1)(6)m_1 = (1, 2, 1)$. However, since the translation $\tau = [1, 5][2, 6][3, 7][4, 8]$ is not defined on $\mathrm{PE}_5 = m_1(t_2)$, we cannot define $[1, 5][2, 6][3, 7][4, 8]$ as a mapping. Formally we can say that the partial permutations $\widehat{(1, 2, 6, 5)}$ and $\widehat{(2, 5)(1)(6)}$ are defined on m_1, but $\widehat{[1, 5][2, 6][3, 7][4, 8]}$ is not defined on m_1.

What happens with application symmetries? As defined here, the edges of the application K are the (data) dependencies of a computation process (or task). All dependencies have to be respected, which means that considering partial symmetries of the application can lead to non-determinstic or faulty behavior.

We are now ready to formally define the set of partial symmetries of architectures and mappings, as in the case of groups. Recall that a mapping $m : K \to A$ can be seen as a morphism of graphs from M to A. In particular, every mapping m defines a subgraph $m(K) \leqslant A$. This subgraph has a node $m(t) \in V_A$ for every PE in the architecture A that is used in a mapping, and similarly an edge $(m(t_1), m(t_2)) \in E_A$ for every communication primitive where a channel is mapped to. Precisely the isomorphism of these subgraphs is what defines the partial symmetries of the architecture.

Definition 4.1.4 (AutSemi)**.** Let A be an architecture graph. The set of partial symmetries of the architecture graph $\mathrm{AutSemi}(A)$ is the set of partial labelled-graph isomorphisms of A, i.e. the partial permutations φ of

V_A which induce an isomorphism of labeled graphs between $\mathrm{dom}(\varphi)$ and $\mathrm{cod}(\varphi)$.

As motivated above, $\mathrm{AutSemi}(A)$ acts on the set of mappings M, just as $\mathrm{Aut}(A)$ the group of (total) symmetries does. This action (and the action of the group $\mathrm{Aut}(K)$) define together an embedding on $\mathrm{AutSemi}(M) \leqslant \mathcal{I}(M)$, the inverse semigroup of partial permutations on M, which is how we define $\mathrm{AutSemi}(M)$.

In inverse semigroups, not every element has an inverse, only a pseudoinverse. Consider the identity partial permutation on the lower-left 2×2-NoC in the 4×4 mesh, $i = (1)(2)(5)(6)$. This identity partial permutation is an *idempotent*, which means that $i^2 = i$, which implies that $i^{-1} = i$. Groups, in contrast, have precisely one idempotent, the identity element. The set of idempotents of a semigroup plays an important role in describing the structure of the semigroup [Law98]. If we then consider the translation τ from above, we can multiply $\tau i = [1,5][2,6]$, which is defined only on two cores. If we muliply it with the pseudoinverse of i, $i^{-1} = i$, we get $\tau i i^{-1} = \tau i i = \tau i \neq \tau$. There is no way we can get τ back from τi, since τ is defined on 4 cores.

Just as with groups, we can define orbits for inverse semigroups. However, due to the one-way nature of some multiplication operation, the orbit of a semigroup is more complicated. Let X be a set and let S be a semigroup acting on X. Then, for an element $x \in X$, we can think of the *orbit graph* of x as a directed graph $\mathbb{O} = (V, E)$ where $V = \{sx \mid s \in S\}$. The edges E are defined by the action, namely an edge $e = (v, w)$ is added for every v, w for which there exists an $s \in S$ such that $v = sw$. This directed graph is clearly connected, but not strongly connected. The strongly connected components (SCCs) of this orbit graph define equivalence classes and play the role that orbits played in group actions for our application to software synthesis.

By definition of $\mathrm{AutSemi}(A)$, for a partial symmetry $f \in \mathrm{AutSemi}(A)$ and a mapping m we know that if the mapping fm is defined, then the two subgraphs $fm(K) \cong m(K) \leqslant A$ are isomorphic. We also get an isomorphism between the two subgraphs by Lemma 4.1.5

Lemma 4.1.5. Let $m : T \to A$ be a mapping and let $f \in \mathrm{AutSemi}(A)$ be a partial automorphism of the architecture such that $\mathrm{im}(m) \subseteq \mathrm{dom}(f)$. Then, the two graphs $m(K)$ and $(fm)(K)$ are isomorphic and the function $\varphi : m(K) \to (fm)(K), m(t) \mapsto f \cdot m(t)$ for all $t \in K$ is an isomorphism of labeled graphs.

Proof. First note that φ is well-defined. Indeed, since $\mathrm{im}(m) \subseteq \mathrm{dom}(f)$ it means that $f \cdot m(t)$ is defined for all $t \in V_K$. Since $f \in \mathrm{AutSemi}(A)$, we know that the type of $m(t)$ and $f \cdot m(t)$ is equal for all $t \in V_K$, as well as the type of all edges $(m(t_1), m(t_2))$ and $(f \cdot m(t_1), f \cdot m(t_2))$ is equal. Thus, φ is a morphism of labeled graphs. Finally, since $f \in \mathrm{AutSemi}(A)$, we know that f is a partial permutation, and in particular, a bijection between $\mathrm{dom}(f)$ and $\mathrm{cod}(f)$. In particular, φ is bijective, and as a bijective morphism of labeled graphs, an isomorphism. ☐

What about the converse, if the subgraphs generated by the mappings are isomorphic, does this mean that there is a (partial) isomorphism of the mappings too? Can we use this to characterize equivalent mappings? In general, no. Consider the subgraph of the mappings $m_2 := (5, 5, 1)$

and $m_3 := (5,1,1)$. Both these mappings project into isomorphic subgraphs $m_2(K) \cong m_3(K) \cong m_1(K)$, but obviously the mappings are not equivalent. Even if the subgraphs are isomorphic, the crucial difference is, however, that the mapping φ as defined in Lemma 4.1.5 is **not** an isomorphism of (labeled) graphs. What if tasks t_1 and t_2 are equivalent? In other words, what if $g = (1,2)(3)$ is a (full) automorphism of the application graph? Then, the mappings m_1 and m_3 are equivalent (via g), but the function φ of Lemma 4.1.5 is still not an isomoprhism of the subgraphs. However, we can generalize the function by applying g first, as $\varphi \circ g : m(T) \to fm(T), m(t) \mapsto (fm \circ g)(t) = (fm)(g(t))$. This generalization, in fact, yields a full characterization of equvialent mappings through isomorphy of subgraphs.

Theorem 4.1.6. Let A be an architecture with inverse semigroup of automorphisms $S = \mathrm{AutSemi}(A)$ and let K be an application graph with group of automorphsims $G = \mathrm{Aut}(K)$. For mappings $m, m' : T \to A$, the following statements are equivalent:

1. There exists a partial permutation $f \in S$ and a permutation $g \in G$, such that $\varphi \circ g$ is an isomorphism of labeled graphs.
2. The two mappings are equivalent by symmetries in the orbit of $S \times G$.

Proof. The implication $(1) \Rightarrow (2)$ follows directly from the definition of φ and the action of $S \times G$. For the implication $(2) \Rightarrow (1)$, since m and m' are in the same SCC of the orbit of $S \times G$, there exists an $x \in S \times G$ such that $m = x \cdot m'$. We can use the direct product structure of $S \times G$ to decompose $x = fg$ for $f \in S, g \in G$. This means that $m = fg \cdot m' = f \cdot (g \cdot m')$. Applying Lemma 4.1.5 on m and $g \cdot m'$ shows that $\varphi \circ g$ is an isomorphism. ☐

How do partial symmetries with inverse semigroups compare to (global) symmetries, in the sense of group theory? We can start with a simple example, of a 2×2 mesh, which we will call M_2. The group of symmetries of this architecture, as we have seen, is D_4 with $|D_4| = 8$ symmetries. What about the partial symmetries? It is easy to check that $|\mathrm{AutSemi}(M_2)| = 45$, which are many more partial symmetries than global ones! But in fact, comparing the size of the group and the semigroup is misleading. We can't compare them, as they deal with different objects, functions and partial functions. For this case of M_2 there is a sense in which we do not get any more symmetries by going to the partial symmetry world. We can see it through the following argument: the group $\mathrm{Aut}(M_2) \cong D_4$ acts canonically on the power set of M_2, $\mathrm{Pow}(M_2)$, simply by acting element-wise: For $M \subseteq M_2$ and $g \in \mathrm{Aut}(M_2)$, the (canonical) action is defined as follows: $g \cdot M := \{g \cdot m \mid m \in M\}$. In this action, the orbits $\mathrm{Pow}(M_2)/G$ are in obvious bijection to the SCCs of the orbit of $M_2/\mathrm{AutSemi}(M_2)$.

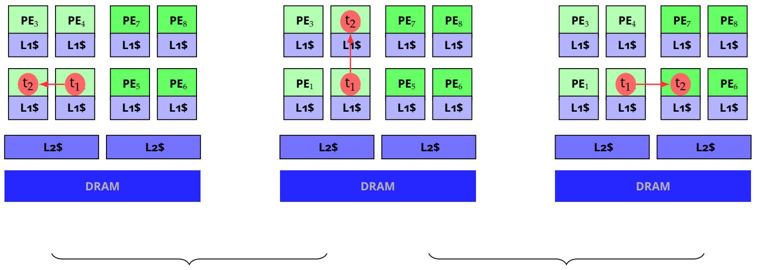
We have seen how to describe partial symmetries, a natural question is how to calculate them? This can be accomplished with the methods of [Eas+19], and our applications of it in joint work with Sergio Siccha and Jeronimo Castrillon [GSC17]. In fact, `mpsym` implements Algorithm 2 from [GSC17]. We worked with Sebastian Krammer in his bachelor thesis [Kra17] on finding more efficient algorithms. Unfortunately, the algorithms as implemented so far are not efficient enough to be useful in the context of mappings and DSE.

In future work, we believe we should be able to find explicit generating systems for an $n \times n$ mesh for an arbitrary n, which would significantly improve the performance of the algorithms, which is limited by finding a good generating set. Using inverse semigroups also opens up an additional avenue for future work, where *similarities* can be described instead of precise symmetries. For example, mapping an edge between two cores in a mesh to a different edge type with a *smaller* number of hops is sure to not worsen the performance of the application when running in isolation, although we cannot say if it will improve it or not. Such a transformation can also be described with semigroups, and the directed graph structure of the orbits nicely encompasses such one-way transformations.

4.2 Metric Spaces

When considering the design space of mappings $M = A^K$ we usually consider no quantitative relationship between mappings. For two mappings we can say if they are identical or not, or perhaps with the methods of Section 4.1 if they are *equivalent* or not. However, any further relationship we can't describe: can we say that two mappings are very similar, or very different? Can we quantify the *distance* between two mappings? Intuitively, we can. This section requires some basic concepts from the mathematic theory of (discrete) metric spaces and embeddings into real spaces. Appendix A.2 gives an overview of the required concepts, a more thorough exposition can be found in [Mat02], Chapter 15 in particular.

Normally, we encode mappings as vectors $m = \left(a_1, \ldots, a_{|V_K|}\right)$ where $a_i \in V_A$ is the PEs where task i is mapped. If we interpret these vectors as being (real) vectors in $\mathbb{R}^{|V_K|}$, we can endow them with a vector distance, like the Euclidean distance $d_{\text{Euclidean}}(v, w) = \sqrt{\sum_i (v_i - w_i)^2}$. This can be generalized to other p-norms, as $d_{L_p}(v, w) = \sum_i ((|v_i - w_i|)^p)^{1/p}$, which is a norm for $p \geqslant 1$. For $p = 1$, this norm is also known as the Mathattan distance, in allusion to the distance between buildings in a regular mesh like the streets of Manhattan. We can endow the space of mappings with a metric also by using the Hamming distance, which counts only the number of differing entries in the vector. However, none of these metrics are ideal for the mapping space, as we will now explain.



$$\text{dist} = \sqrt{(4-1)^2} = 3 \qquad \text{dist} = \sqrt{(4-5)^2} = 1$$

Figure 4.10: An intuitive example of distance between mappings.

Consider the example in Figure 4.10. It shows three mappings

$$m_1 : t_1 \mapsto \mathrm{PE}_2, t_2 \mapsto \mathrm{PE}_1; \quad m_2 : t_1 \mapsto \mathrm{PE}_2, t_2 \mapsto \mathrm{PE}_4;$$
$$m_3 : t_1 \mapsto \mathrm{PE}_2, t_2 \mapsto \mathrm{PE}_5 \,.$$

We would normally write these mappings as vectors, $m_1 = (2,1), m_2 = (2,4)$ and $m_2 = (2,5)$. If we calculate the standard (Euclidean) distance of these vectors, then m_2 is farther away from m_1 than from m_3. However, we know that communication between PE_1 and PE_4 is much faster than between PE_4 and PE_5. The Euclidean distance in the mapping space does not reflect the structure of the communication subsystem.

4.2.1 *Architectures*

In the example illustrated in Figure 4.10 we saw intuitively how mappings can be more or less similar. This intuitive notion clearly depends on the underlying architecture. It is the hardware architecture that determines the cost of communicating data between processes. In order to endow the space of mappings with a metric space structure, we should first do so with the architecture.

We can use the intuition behind the example to define a metric that takes latency into account this way [GMC18]. The fundamental observation here is that in a multicore architecture, communication between different PEs takes different amounts of time. There are multiple problems with using the communication time between PEs directly as a distance between PEs. Firstly, communication times depend on multiple factors: the latency and bandwidth of the communication resources used, the amount of data being sent, the (software) communication protocol, clock synchronization between hardware resources like the PEs and buses, arbitration or other contention issues, etc. Of course, we can model these to various degrees. However, the distance between PEs needs to be a fixed number and not a function of all these factors. As an approximation, however, we can use the expected latency for a package of a standardized size (e.g. 8 bytes). As an expected value, this is a fixed number, but through its statistical nature it can include as much complexity in the model as required[2].

The second issue we run into when using communication times for defining a distance is that, by definition, the distance between a point and itself has to be 0, but usually a PE has to communicate with itself using an $L1$ cache, scratchpad memory or similar, which has a small but non-zero latency. In this sense, the expected communication latency between cores is **not** a metric space distance, but it approximates one well. We propose thus to ignore this latency and set the distance to 0, to obtain the mathematical metric space structure.

Finally, this metric space structure depends strongly on the unit used to measure latency (e.g. cycles, milliseconds, etc), as well as on the absolute speed of the communication sub-architecture. Since the goal of exposing this structure is to leverage it for algorithmic decisions like finding good mappings, it is useful to have comparable distances between different

---

[2] If communication in the architecture is asymmetric, this will not define a metric. We can average the communication from p to q and from q to p to fix this, but we should probably consider this case separately.

architectures. For this, we propose to norm the metric distance function such that the average distance between PEs is 1.

Put together, these principles yield the following definition:

Definition 4.2.1 (Architecture Metric Space)**.** Let $A = (P, E)$ be an architecture graph and $\mathrm{lat} : P \to P$ be the expected latency between PEs. Then we set

$$d_A : P \times P, (p, q) \mapsto \begin{cases} \mathrm{lat}(p, q), & \text{if } p \neq q \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

Remark 4.2.2. For an architecture graph $A = (P, E)$, the tuple (P, d_A) is a metric space.

Proof. Obviously $d_A(p, p) = 0$ for all $p \in P$, by definition, and $d_A(p, q) > 0$ for $p \neq q$ since the expected latency between PEs is always greater than 0. For $p, q, r \in P$ we have $d_A(p, q) + d_A(q, r) \geqslant d_A(p, r)$ since the expected latency of moving data from p to q and then to r will always be at least as much as moving it from p to r directly. ☐

In this way we endow M with a discrete metric space structure, with a metric that reflects the memory subsystem of the architecture, or more generally, its communication. While this allows for a simple and powerful mathematical definition, a metric space structure can be inefficient for calculations. To cope with this, we will also discuss low-distortion embeddings and show how we can find them for the metric spaces introduced. Appendix A.2 reviews the basic notions of metric spaces, as well as more advanced concepts needed to introduce and find the more computationally efficient low-distortion embeddings.

Unfortunately, this metric also has some issues. In particular, it does not distinguish between core types on heterogeneous systems. To fix this, we propose an alternative metric space structure on M, by adding extra dimensions for the communication and the computation. This is fundamentally very similar to adding channels in the mapping vectors. We thus define a metric on the channels, based on the metric defined by Definition 4.2.1. The distance between two channels $c, c' \in E_A$ is defined as $|\mathrm{lat}(c_1) - \mathrm{lat}(c_2)|$ for the communication channel between the cores. We then apply a similar concept for the cores, and take relative values of the expected runtime. Disregarding the ISA or micro-architecture, we can use the frequencies as a first estimation, which is what we do here. Obviously the frequency is not the best estimation of the expected differences in execution times between PEs, but we restrict our consideration to this for the scope of this thesis. Future work should focus on finding better metrics for the mapping space.

This definition will not produce a metric, since distinct cores which are equivalent will have a distance of 0, and similarly equivalent channels. To deal with this, we add a minimal distance between the cores (e.g. 0.1 times the distance between the next two core types).

Application distances

To go from A to M, we can use the same principle as the L_p norms and define $d(m, m') = (\sum_i d(m_i, m'_i)^p)^{1/p}$, which can immediately be checked to be a metric on M. This way we can consider, as a metric space (embedding), the structure of A to be

$$M \underbrace{\perp \ldots \perp}_{\times|V_K|} M, \text{ i.e. } M \underbrace{\times \ldots \times}_{\times|V_K|} \times M \text{ with } d(M_i, M_j) = \{0\} \text{ for all } i \neq j. \tag{4.2}$$

There are multiple issues with this as well. A crucial problem with it is that this does not consider the dependencies between tasks in the application graph A, nor does it consider how multiple tasks might be more or less relevant. Many methods can be considered to account for this fact, like having factors for the dimensions of the copies of M in the orthogonal sum. However, we omit evaluating multiple such metrics to limit the scope of this thesis.

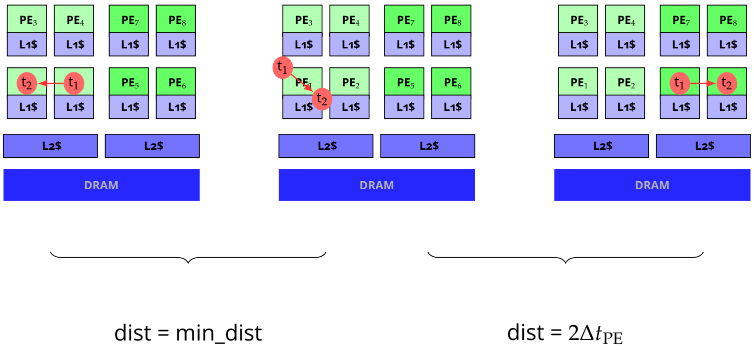


Figure 4.11: An example of a problem with the orthogonal-sum construction of the distance metric for the mapping space.

Figure 4.11 illustrates another problem with this construction. The metric does not distinguish between tasks mapped on the same core or on different cores, something that has a large impact on the performance of the mapping. Here we are considering the variant of the metrics with the extra dimensions, but the problem is independent of the architecture metric we base this on.

A particularly important property of these metric space constructions is that they give meaning to distances in the mapping space; they make it into a landscape. This is a highly-dimensional landscape, which we cannot visualize except in the simplest examples, like the two-task mapping we visualized previously, in Figure 1.4. There are other ways of visualizing this space, however. The t-SNE method [MH08] aims to group points by their distances, making points that are close by in the mapping space also close in the visualization, and simlarly for points far appart. A disadvantage of this method is that it does not preserve the actual values, the coordinates of the points become meaningless. A different approach is to use random projections onto a two dimensional-space. By the Johnson-Lindenstrauss lemma [JL84], such a random projection will have a low distortion with high probability (see Appendix A.2 for more details). We use the methods of [Li+18a] to visualize such a random projection of the mapping landscape.

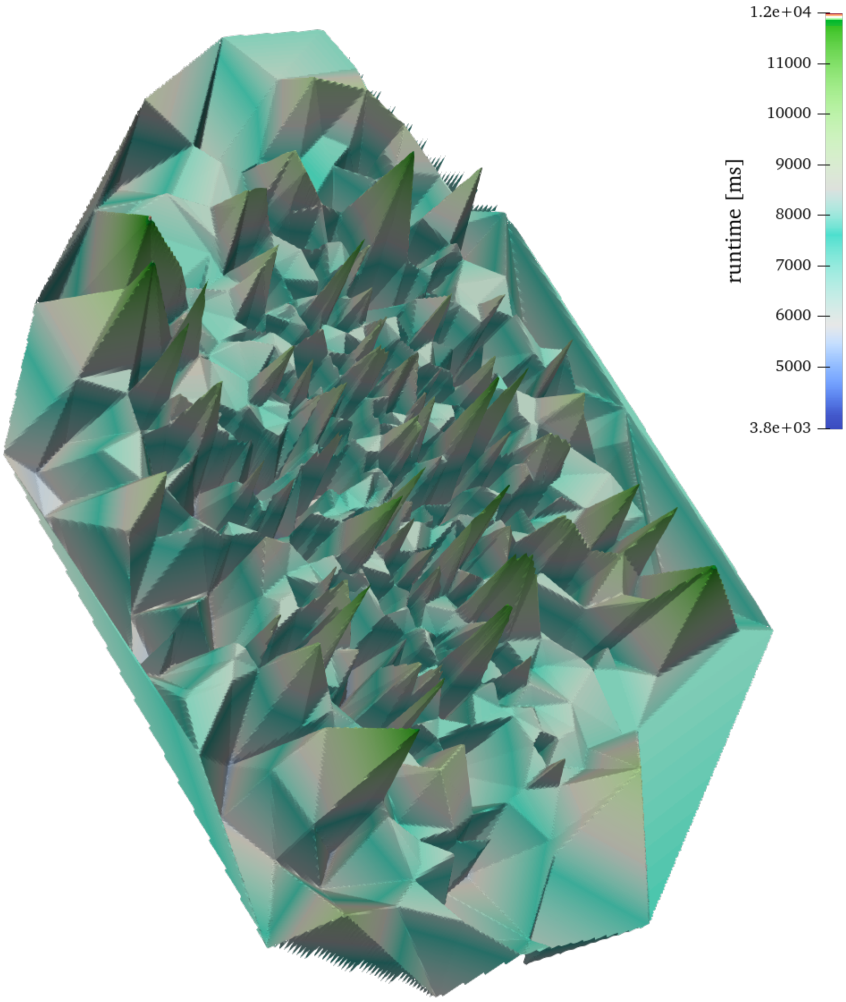


Figure 4.12: A visualization of a random projection of the mapping space

4.2.2 *Mappings*

Figure 4.12 shows a rendering of the design space of mappings for the audio filter CPN benchmark onto the MPPA3 coolidge. We generate this rendering using the methods of [Li+18a], generating a smoothening from a triangulation of a random projection of 1000 random mappings as an artistic interpretation that we can visualize with ParaView [AGL05]. This was generated from the Euclidean norm on the mapping space interpreted as vectors. The height of the mountains and valleys in this landscape, as well as their coloring, represent the value of the execution time for the mappings being visualized. We see how the mapping space has multiple local minima and maxima, a fact which we will discuss more in Section 5.3.

4.2.3 *Low-distortion Embeddings*

We have seen so far how we can endow the mapping space with multiple metrics $d_M : M \times M \rightarrow \mathbb{R}_{\geqslant 0}$ to define distances between mappings. A problem with this is that the mapping space is a discrete space, with a very large cardinality. To algorithmically do any computation in this space, e.g. in DSE, we need to iterate through the whole space. For example we might have a mapping m_0, for which we want to find all mappings that are within a radius r of it, i.e. compute the ball $B_r(m_0)$ with radius r around m_0. For this we need to iterate over all $m \in M$ and calculate if $d_M(m_0, m) \leqslant r$, which is intractable for all but the simplest examples.

To deal with this, we use established methods from discrete geometry to calculate *low-distortion embeddings*. A mapping $\iota : M \hookrightarrow \mathbb{R}^n$ such that there exists a $D > 0$ with

$$D^{-1} d(x,y) \leqslant \|\iota(x) - \iota(y)\| \leqslant d(x,y) \tag{4.3}$$

is called an embedding with distortion D (cf. Appendix A.2). In other words, the *relative error* of the distances is at most D. Using convex optimization [Mato2], we can calculate a low-distortion embedding for a finite metric space. This allows us to work with vectors of real numbers which make many algorithmic tasks scalable, e.g. computing random points in a ball.

Since the size of the mapping space grows exponentially with the number of tasks and changes for every application, computing such an embedding for a large mapping space every time we want to do DSE would also be intractable. We can avoid this by using the orthogonal sum construction from Equation 4.2. Given an embedding $\iota : A \hookrightarrow \mathbb{R}^k$ with distortion D for the architecture with a given metric d_A, we can construct an embedding ι^k of the mapping space defined as in Equation 4.2 with distortion D.

Theorem 4.2.3 (Theorem III.1 of [GMC18])**.** Let $\iota : (\mathcal{M}, d) \hookrightarrow (\mathbb{R}^n, \|\cdot\|_p)$ be an embedding with distortion D and define $\iota^k : (\mathcal{M}^k, d_p) \hookrightarrow (\mathbb{R}^{nk}, \|\cdot\|_p)$ as $\iota^k((x_1, \ldots, x_k)) = (\iota(x_1), \ldots, \iota(x_k))$. Then ι^k is an embedding with distortion of at most D.

Proof. It is clear why ι^k is an embedding (well-defined and injective), since ι is one. The distortion follows from the homogeneity of the $\|\cdot\|_p$-norm applied to Equation 4.3. ☐

The mapping space can still have a high dimension, a problem usually called the *curse of dimensionality*. With this construction, for the metric without the extra dimensions, the dimension of the embedding ι^k is $k|V_A| = |V_K||V_A|$. A method to improve this is to use the Johnson-Lindenstrauss lemma to reduce the dimension with a projection. We do this with an iterative method, described in Algorithm 3.

Algorithm 3 Iterative dimensionality reduction via the Johnson-Lindenstrauss lemma.

```
input: A discrete metric space $M$, a low-distortion embedding $\iota : M \hookrightarrow \mathbb{R}^n$
       and a target distortion $D$.
output: An embedding with dimension $\leqslant n$ and distortion at most $D$.
dim ← 1
while dim $\leqslant n$ do
    for _ ∈ numIterationsPerDim do
        $\tilde{\iota} \leftarrow \mathrm{JLReduction}(\iota, \mathrm{dim})$
        $\tilde{D} \leftarrow \mathrm{CalculateDistortion}(\tilde{\iota})$
        if $\tilde{D} \leqslant D$ then return $\tilde{\iota}$
    dim ← 2 · dim
return $\iota$
```

Algorithm 3 exponentially increases the dimension, running `numIterationPerDim` iterations of a Johnson-Lindenstrauss transform and testing the distortion to see if a target distortion has been reached. Using this algorithm, or variants thereof, we can control the trade-off between the distance and the dimension of the embedding.

We can combine these embeddings with symmetries, by first calculating a canonical representative (cf. Problem 3 of Section 4.1) and then calculating the embedding using $\tilde{\iota}^k$ as defined here. We used these methods to evaluate and compare multiple metrics. A useful property of this metric would be to have the distance between two mappings correlate with the (relative) relation of their runtimes. In other words, we would expect a good metric to be larger for mappings that have very different performance and lower when mappings have similar performances.

To compare the different metrics and embeddings, for each of them we calculated 1000 mappings of the audio filter benchmark on the Odroid XU4 platform. For a random subset of the $1000^2 = 10^6$ pairs of mappings we calculated the (relative) distance between two mappings and the relative simulated runtime of these two mappings.

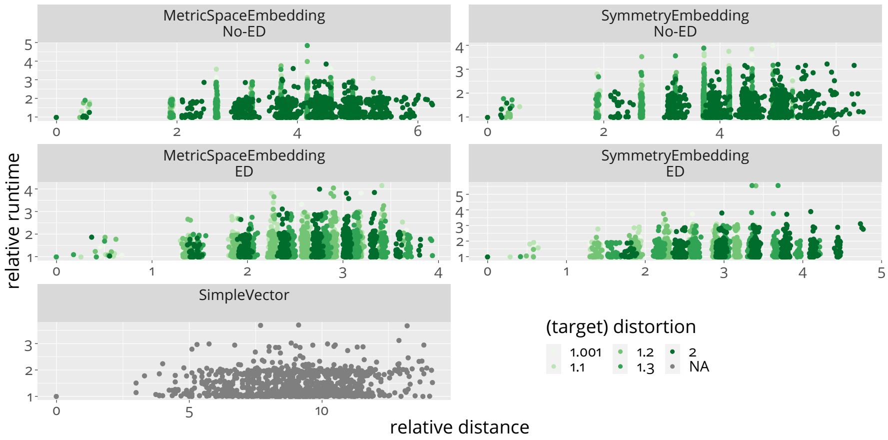


Figure 4.13: Comparison of multiple distance metrics on the Odroid XU4 platform.

Figure 4.13 shows the comparison of the different metrics and embeddings. In the figure, the metrics described in this section are labeled as follows: We call `SimpleVector` the Euclidean norm on the mappings described as simple vectors. The metric based on the latencies as motivated from Figure 4.10 we denote as `MetricSpaceEmedding`, whereas we add the annotation `ED` for the metric with extra dimensions which accounts for heterogeneous PEs. The variants described as `SymmetryEmbedding` and `SymmetryEmbedding ED` are the two respective metrics combined with the symmetries, as described above. These names and details of how we included the symmetries will be explained in more detail in Section 4.3.

The results from the figure show that there is basically no correlation between mappings distance and the (relative) runtimes. Two mappings can be very far apart and have (almost) the same execution time. This seems very plausible if we consider the symmetries of the problem. The symmetry reduction as used in the figure, for example did not consider the application symmetries (cf. Section 4.1). Moreover, some *similarities* are also not considered. The computation for the FFT and inverse FFT (IFFT) in the audio filter benchmark is virtually identical, yet not precisely identical, and would not be captured by application symmetries either. In practice, this leads to very similar if not identical mapping execution times nevertheless.

A perhaps better assessment of the metrics is to ask what is the *maximal* relative execution time possible for a given distance. While we understand why two similar mappings that are far apart will have similar results, we would expect two mappings that are close to each other to have similar execution times with a good metric. To test this, we take the same results of Figure 4.13 and just consider the maximal relative execution time for two mappings which are (at most) the given distance appart.

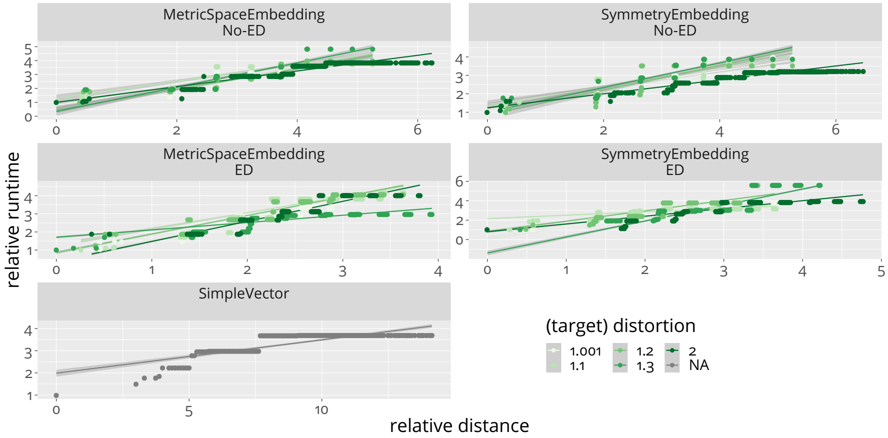


Figure 4.14: Comparison of multiple distance metrics as predictors of the *maximal* run-time difference on the Odroid XU4 platform.

Figure 4.14 shows this maximal relative execution times for the data of Figure 4.13. It also includes a linear regression of the points for each metric and embedding. We can see that indeed, most of the metrics are pretty good as a *bound* on the relative runtime.

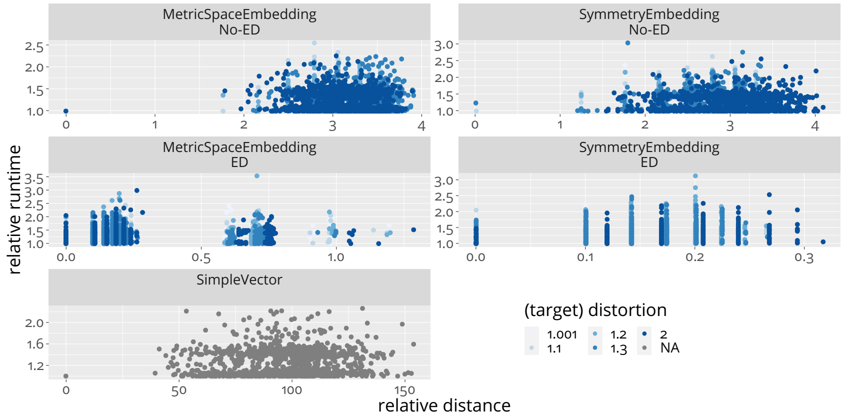


Figure 4.15: Comparison of multiple distance metrics on the MPPA3 Coolidge platform.

The Odroid XU4 architecture is comparatively small, which obviously has consequences for the mapping space. There are, for example, fewer problems with the curse of dimensionality and overall a smaller (discrete) space. How does the situation change for the MPPA3 Coolidge? Figure 4.15 shows the same comparison as above for the MPPA3 Coolidge. We can see that the space is more complex. In particular, the extra dimensions clearly make a difference, separating the space into more discrete distinct points, with clearly visible vertical lines. We will discuss these vertical lines further in Section 5.3.

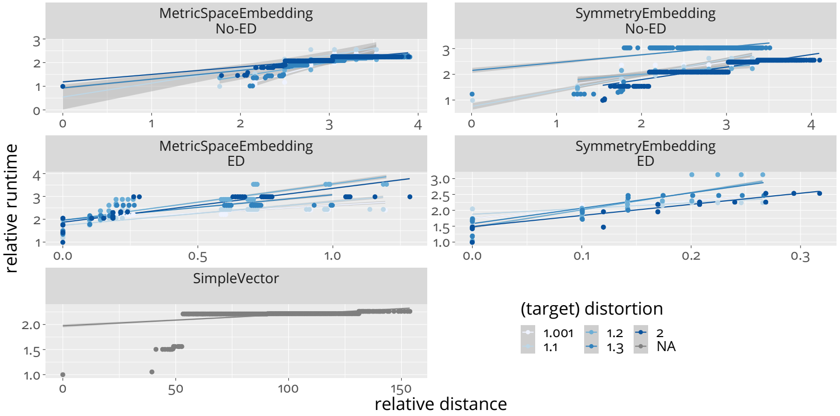


Figure 4.16: Comparison of multiple distance metrics as predictors of the *maximal* run-time difference on the MPPA3 Coolidge platform.

Similar to the case for the Odroid XU4, Figure 4.16 shows the same comparison with the maximal run-time difference for the MPPA3 Coolidge. Again we see that many metrics seem to be a decent bound for the difference in execution time, although less so than for the simple Odroid XU4 platform. The Euclidean norm on the simple vector mappings, for example, is considerably worse than in this case than in the Odroid XU4. We can quantify more precisely how good metrics are as a bound for the execution time by comparing the R^2 value as goodness of fit assessment of the depicted linear regressions.

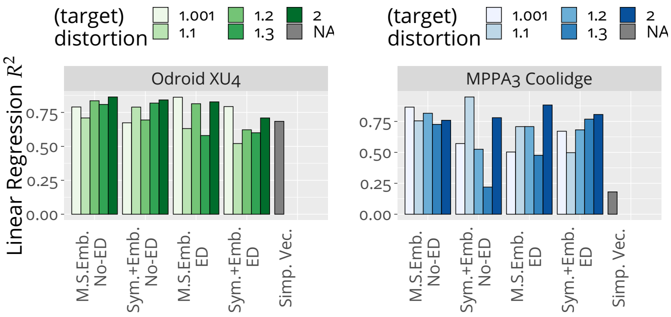


Figure 4.17: Comparison of the predictive power of multiple distance metrics.

Figure 4.17 shows the R^2 value, comparing the predictive power of the different distance metrics and their embeddings. Here it is also very clear that the Euclidean norm on simple vectors is not so good for the MPPA3 Cooldige, while it is comparable to other metrics in the Odroid XU4. We also see how the curse of dimensionality yields a trade-off not only in the computation time (for larger-dimensional spaces), but also in the predictive quality of the different norms. This is more visible on the MPPA3 Coolidge.

When working with mappings, particularly for DSE, we normally want to represent these mappings in different ways for the algorithms [GMC18]. In particular, we usually work with mappings using vectors to represent them. In most cases and flows, this is done implicitly, usually with the simple vectors we have been describing in many places so far. A mapping $m : K \to A$ is described as a vector $(m(v_1), \dots, m(v_k)) \in V_A^k$, where V_A is interpreted to be a subset of $\mathbb{Z}$, assigning integer values for the different PEs in V_A. Sometimes this representation is extended to consider channels, $(m(v_1), \dots, m(v_k), m(e_1), \dots, m(e_l)) \in (V_A \cup E_A)^{k_1+k_2}$, adding more integers to represent the communication primitives. In Section 4.2 we saw how we can interpret this as a metric space by using the Euclidean distance. This metric space structure is also commonly assumed when adapting meta-heuristics from other domains for mapping, often without giving the metric space structure any second thought.

In this section we want to generalize this representation and use the concepts introduced in this chapter to define three additional such representations. Here we define a *representation* to be specifically an embedding of the mapping space to a (real) vector space, for manipulating mappings e.g. in meta-heuristics for DSE. We refer to the simple vector representation we just discussed as the `SimpleVector` representation. In `mocasin` we implement representations as a type of "lens" to see mappings in different ways. We used this representation implicitly for example in Figure 1.4, where we described the mapping space of a two-task application to the Odroid XU4, which as a two-dimensional space in the `SimpleVector` representation so that we could visualize the mapping space on a plot.

In Section 4.1 we introduced symmetries of the mapping space, and explained how equivalent mappings in an orbit have the same objective values Θ, like run-time, at least in simulations. We can define a (vector) representation to factor out the symmetries, as described when discussing Problem 3 in Section 4.1. We do this by choosing a canonical representative for every orbit, specifically the minimal element of the orbit with regard to the lexicographical ordering. Thus, in `mocasin`, when "inspecting" a mapping with the `Symmetries` representation, it returns the lex-minimal element in that mapping's orbit. This effectively prunes the design space, factoring out the symmetries.

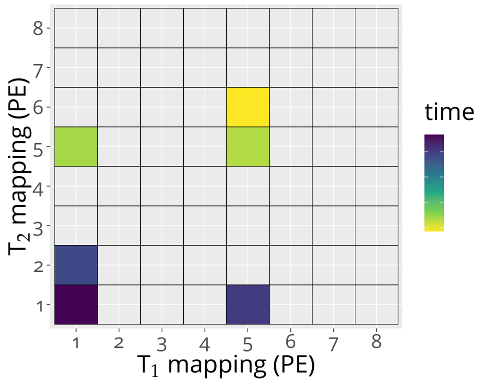


Figure 4.18: A visualization of the mapping space of Figure 2.9 in the `Symmetries` representation.

Figure 4.18 shows the same two-task application from Figure 2.9, this time seen through the `Symmetries` representation. Only the lex-minimal elements remain, which for this simple example are exactly 6 mappings, namely $(1,1),(5,1),(1,2),(1,5),(5,5)$ and $(5,6)$. From this figure it is intuitively clear why this representation prunes the mapping space, as well as why this might be useful for DSE.

We also implemented the low-distortion embeddings discussed in Section 4.2 in `mocasin` as the `MetricSpaceEmbedding` representation. We implemented both additional distance metrics described in Section 4.2 and enable the one with extra dimensions by default. By default, we set a target distortion of 1.001, which effectively disables the Johnson-Lindenstrauss reduction. We do this because it is yet unclear how to best manage the trade-off enabled by this transform, to define a sensible default, and we want to keep the distortion of the metrics as close to the defined metric as possible. Combining both representations, with the embeddings and the symmetries, we obtain the `SymmetryEmbedding` representation. A mapping inspected through this representation is first normalized to the lex-minimal element using symmetries and then embedded into a real space using the low-distortion embedding.

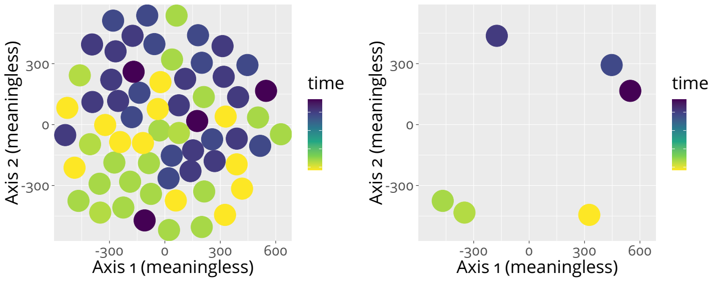


Figure 4.19: A visualization of the mapping space of Figure 1.4 in the `MetricSpaceEmbedding` (left) and `SymmetryEmbedding` (right) representations.

Figure 4.19 shows an intuitive visualization of `MetricSpaceEmbedding` and `SymmetryEmbedding`, the two representations based on low-distortion embeddings, on the same two-task application example on the Odroid XU4. The real vector spaces to which we embed these spaces are in this case 30-dimensional, not 2-dimensional. To visualize them we use the t-SNE [MH08] method, which is not a good representation of the actual distances but provides a good overview of the structure of the whole space. We see how in these metrics the structure of times is intuitively better organized than in the `SimpleVector` representation with the Euclidean norm.

Finally, Figure 4.20 gives an overview of the four representations of the mapping space introduced here on the example of the two-task application mapped onto the Odroid XU4.

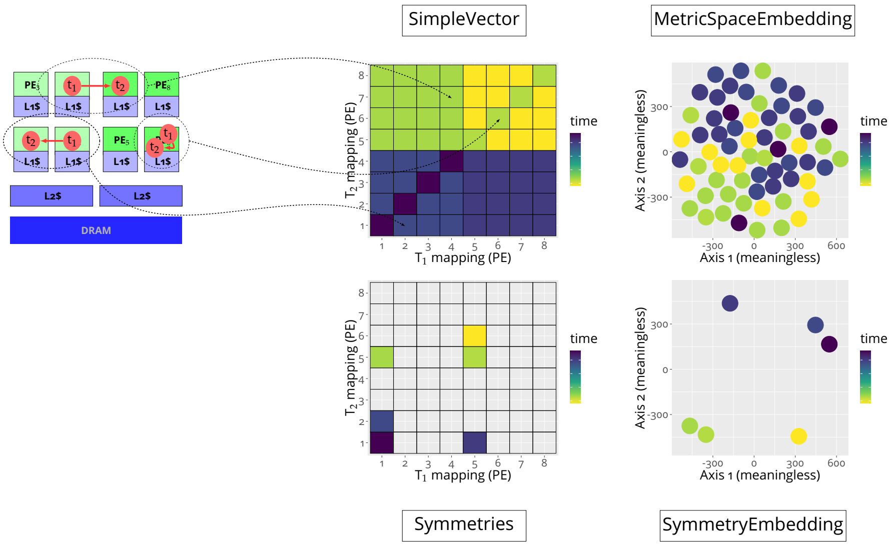


Figure 4.20: An overview of all four representations discussed in the example of the mapping space for a simple two-task application from Figure 1.4.

5 APPLICATIONS OF MATHEMATICAL STRUCTURES IN MAPPINGS

In Chapter 4 we have seen how the mapping space has inherent structure and how we can describe this structure explicitly using mathematical methods. In this chapter we will see how we can leverage this structure to improve software synthesis flows in different ways.

5.1 Compact Mappings

In Section 4.2 we show multiple ways of endowing the mapping space with a distance metric. A common method for defining a metric in a NoC-based system is to count the number of hops between two processors [Sin+10; Sch+17]. Indeed, this is the same as the L_1 (Manhattan) distance on the topology graph of the architecture. A natural idea that arises from this is to search for *compact* mappings, i.e. mappings that take a (geometrically) small area in the chip.

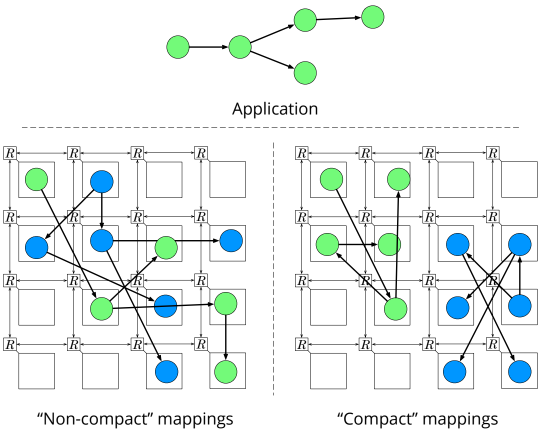


Figure 5.1: Equivalent mappings of two applications, one being compact and the other one not. Adapted from Figure 1 in [GMC19].

Figure 5.1 illustrates the idea of compact mappings. It depicts two variants for mapping the two application graphs depicted in the figure. The particular property of these two variants is that they are equivalent from the point of view of the distances: For any two connected nodes in any of the application graphs, the node distance in terms of number of hops between both nodes is identical in both mapping variants. Intuitively, however, the mappings on the right are preferable to those on the left. Does this intuition translate to actual benefits in mappings?

We first have to translate this intuition into a formal definition. We define the *support* of a mapping m as the set of cores that have tasks mapped to them, i.e. $\operatorname{supp}(m) = m(V_K) \subseteq V_A$. We can look at the size of the support in the metric interpretation of the mapping space. Let r_m be the minimal radius $r > 0$ such that there is a ball $B_r(v_0)$ with radius r for a point $v_0 \in V_A$ that contains the support of the mapping, i.e. $\operatorname{supp}(m) \subseteq B_r(v_0)$. A compact mapping is a mapping with a small r. How small r should be to be considered compact, depends on factors like the metric space and the number of tasks. What we can do properly is compare r_m for different mappings to see if they are more or less compact, according to this definition. For the examples in Figure 5.1 we can see that both mappings on the left have a radius of $r_m = 3$ according to the L_1 (Manhattan) distance, whereas those on the right both have a smaller $r_m = 2$.

To test this idea we used a SystemC-based NoC simulator, Noxim [Cat+15], which we modified to obtain more detailed statistics about the simulations [GMC19]. In particular, we extracted the variance of the package delays in the simulation. We configured Noxim to simulate a 10×10 mesh topology with xy routing and worm-hole switching. This choice was made to mimic the routing of commercial platforms like the Tile-Gx series from Mellanox Technologies [Mel15a; Mel15b], Intel's Xeon Phi [Tam+18] Scalable Platform [Sod+16], or academic ones like OpenPiton [Bal+16].

If we execute the example from Figure 5.1, the non-compact example on the left actually outperforms the compact one on the right. By closer inspection of the figure, this is because the distances within the application are very high. In other words, the mappings depicted are simply bad mappings. A lot of contention within the application offsets any gains from avoiding contention against other mappings.

However, while the motivational example is not very informative in terms of finding good mappings to combine, it does motivate the idea of compact mappings. We used a heuristic to find such compact mappings in a regular mesh NoC, while also ensuring they are not as bad as those in the example. We do this by ensuring the communication costs are low within the application as well, using a greedy heuristic.

Algorithm 4 A greedy heuristic for low-communication mapping in NoC-based architectures. Adapted from Algorithm 1 in [GMC19].

```
input: A connected application graph $K = (V_K, E_K)$, the size of the mesh
    $n$, a set of occupied cores $X \subseteq \{1, \dots, n\} \times \{1 \dots, n\} =: V_A$
output: A mapping $m : V_K \rightarrow V_A$
CurNode ← RandomFrom($V_A \backslash X$)
$v_0 \leftarrow \mathrm{Root}(K)$
mapping ← ($v_0 \mapsto$ CurNode)
$X \leftarrow X \cup \{$CurNode$\}$
for $e = (n_1, n_2) \in \mathrm{BreadthFirstEdgeSearch}(K)$ do
    CurNode ← mapping($n_1$)
    $d \leftarrow \min_{d=1\dots n}\{a \in V_A \backslash X \mid |a -$ CurNode$| \leqslant d\} \neq \emptyset$
    $q \leftarrow \mathrm{RandomFrom}(\{a \in V_A \backslash X \mid |a -$ CurNode$| \leqslant d\})$
    mapping($n_2$) ← $q$
    $X \leftarrow X \cup \{q\}$
return mapping
```

The heuristic is described in Algorithm 4. We assume the application graph is (weakly) connected. The heuristic then starts with any node in the application such that there is a path from it to every node in the application (`Root`). It then randomly assigns an unused core to this node, subsequently iterating through the application graph in a breadth first fashion. In this breadth-first search, it assigns cores such that the distance from a node to its predecessor is minimized in the mapping. This greedy algorithm thus minimizes local communication, but it does not ensure that the communication is minimized globally for the whole application.

A central concept behind this algorithm is the set of cores marked as occupied, which can be initialized in a particular fashion to enforce the geometry of the mapping. To produce compact mappings we mark every core as occupied, except for an $m \times m'$ rectangle such that $m, m' > |V|$, and we choose $\{m, m'\} \subseteq \{\sqrt{|V|}, \sqrt{|V|} + 1\}$ minimal with this property. We compare these mappings to those produced without the additional rectangle constraint, which are low-communication mappings that are not necessarily compact.

To evaluate the concept of compact mappings we again used the Noxim simulation. We generated random task graphs with a Gilbert random graph approach (cf. Section 3.3) with 10 random applications with a variable number of tasks (4-6). We simulated all 10 applications running together in the system, multiple times, using different mapping strategies. For each application we then generated 100 random, 100 compact and 100 non-compact (low-communication) mappings. We tested using a fixed packet size of 32 flits (although the results were similar with packet sizes of up to $2^{12} = 4096$ flits). Packages in Noxim were injected with a fixed injection rate (of packages per cycle), which we also varied between 10^{-2} and 10^{-5}.

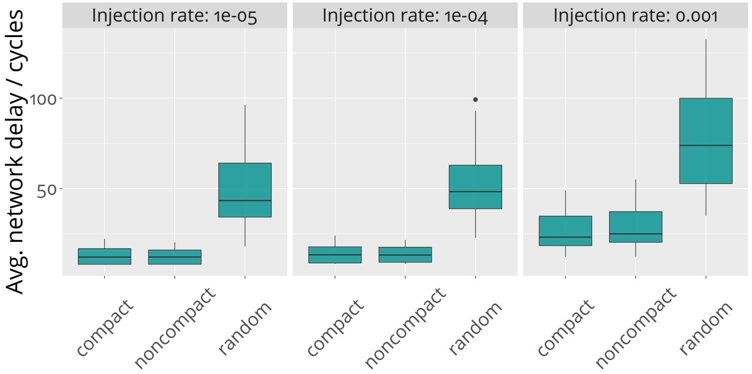


Figure 5.2: Comparison of latencies between compact, non-compact and random mappings. Adapted from Figure 2 in [GMC19].

Figure 5.2 shows the results of a comparison between compact, non-compact and random mappings. Each point reports the *average network delays* over the course of the whole simulation for each of application and each mappings. We can see that for injection rates above 10^{-3} there is basically no difference between compact and non-compact mappings, and for the very high injection rate of 10^{-3} the difference is still negligible.

To make a better comparison, we designed an additional experiment that compared between two distinct scenarios. In one scenario, the application was running alone in the system (isolated), and another one where it was executing alongside 9 additional applications (joint). In the case of the joint applications we report the values of one specific application (the same as in the isolated case), ignoring the rest. The other applications only serve to create contention.

Figure 5.3 shows the results of this experiment comapring applications running in isolation and with contention. For an injection rate of 10^{-5} contention is so low that it makes no difference between two cases (except for random mappings). When increasing the rate to 10^{-4}, while contention

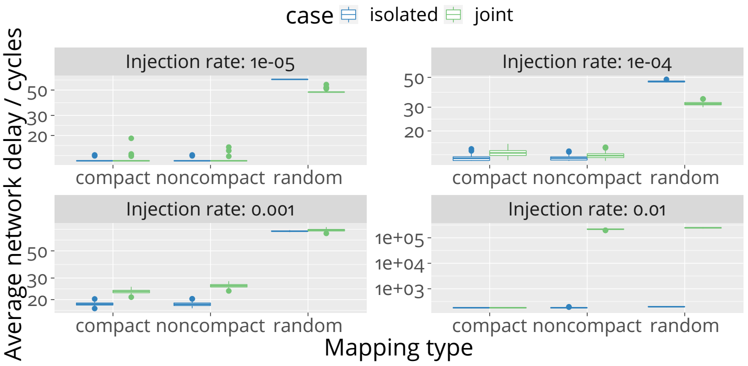


Figure 5.3: Comparison between compact, non-compact and random mappings running isolation or with another 9 applications. Adapted from Figure 4 in [GMC19].

does affect applications, the compactness of the mappings still makes no difference. Compact mappings do perform slightly better only for very high injection rates (10^{-3}) and significantly better for extremely high injection rates 10^{-2}. Note that an injection rate of 10^{-2} every task is sending a package in average every 100 cycles, which is arguably of no practical relevance.

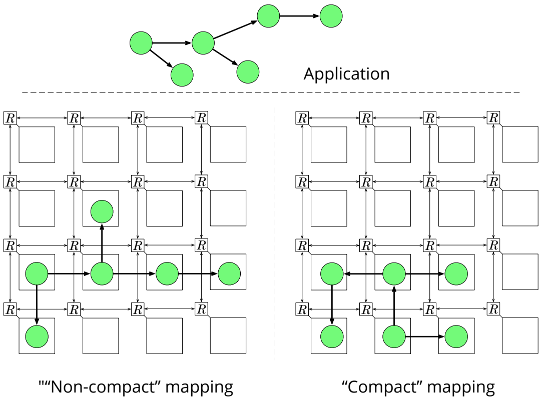


Figure 5.4: Two equivalent mappings that yield good performance. Adapted from Figure 5 in [GMC19].

From these experiments we can conclude that compact mappings are not particularly useful. The improvements they bring in reducing contention are not relevant in practice, since the amount of contention required for the compactness to make a difference is unrealistically high. Figure 5.4 shows a comparison between a typical example of two low-communication mappings, a compact one and a non-compact one. These are the kinds of mappings that result from Algorithm 4 (with and without the additional compactness constraint). These two mappings are good in the sense that they produce low average network delays, as they minimize the distance that the tokens travel and the contention they generate. While their geometries are very different, one being compact and the other one not, their topologies are identical. These two mappings are also equivalent in the same sense as the mappings in Figure 5.1 were. The number of hops between any two nodes is identical for both. A key takeaway thus is that the topology of the mapping is much more important than its geometry, as seen by comparing the compact mapping in Figure 5.1 with the non-compact one in Figure 5.4 (note that the application is sligthly different).

5.2 Robust Mappings

Faulty cores are an unfortunate reality of MPSoCs. After some time, at least one core is likely to fail. However, using hardware monitors, these faults can be reliably detected, sometimes even before the core actually starts failing [ZK11; Zha19]. A strategy to deal with faulty cores, when detected, is to migrate tasks executing in that core to a different core. This way, when the core fails, the execution can continue without the application failing.

While such a remapping strategy is ideal for preserving the functional correctness of applications, it can have negative consequences on the performance of the application. Especially for real-time applications, where the timing performance is part of the functionality, these consequences can be as fatal as a core failing without being detected or without remapping. Moreover, in mixed-criticality domains, a pre-determined mapping can be varied at runtime due to priority issues or similar unforseen circumstances. To deal with this, we propose to search for mappings that are *robust* [Hem+17]. We say a mapping is robust when its runtime properties are unchanged by minor variations in the mapping.

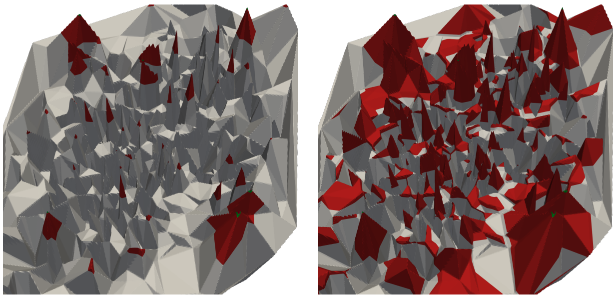
The robustness of a mapping and the corresponding methods proposed in this section are appropriate for soft or firm[1] real-time applications, especially in mixed-criticality contexts. In this context, we say a mapping is *feasible* if its execution time is below a specified real-time deadline. To test if a (feasible) mapping is robust, we apply *perturbations*. A perturbation consists in taking the mapping and changing it partially, to see if it is (still) feasible. A robust mapping should be resistant to perturbations, as motivated by the remapping scenarios described before.

To find such robust mappings we propose [Hem+17] adapting the bio-inspired algorithm for called L_p-adaptation [AMS17]. This algorithm uses the metric space structure of the mapping space (cf. Section 4.2) to navigate it and find a *design center*. For a fixed probability P, a design center is a feasible point m in the design space, such that points in a neighborhood of m are feasible with probability at least P. For the context of this discussion, we consider neighborhoods of the form $B_r(m)$, a ball with radius r around the point m. The L_p-adaptation algorithm seeks to find an m which maximizes the radius r such that the $B_r(m)$ is feasible with probability at least P.

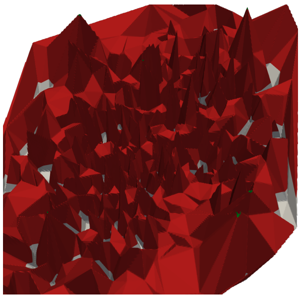
Figure 5.5 again uses a visualization with the method described in [Li+18a] to illustrate the intuition behind the design centering algorithm. We see the mountain landscape of the mapping space, where the height is the execution time of that mapping. The figure shows three different possible thresholds (high, medium, low). All peaks that are above the threshold are colored red: these depict infeasible points. A robust mapping is a mapping such that we could “walk” in any direction from it without reaching one of the red peaks. The larger the radius where this is possible, the more robust the mapping. This metric interpretation of the design space allows us to use the metric-based algorithm of L_p adaptation to estimate

---

[1] A firm real-time application is one where the computation and data is useless after missing a deadline, yet a small percentage of missed deadlines might still be tolerable.



(a) High threshold.
(b) Med. threshold.



(c) Low threshold.

Figure 5.5: Visualization of the design space for multiple thresholds

these neighborhoods around which we can walk without reaching a red peak.

The particular principle behind the L_p-adaptation algorithm is that it is based on estimation using an L_p ball, i.e. with the norm $||x||_p = (\sum_{i=1}^{n} x_i^p)^{1/p}$. The algorithm is inspired on the evolution of robustness in biological systems [AMS17]. The basic principle behind the mapping is a two-step process, where we simultaneously search for a design center and estimate its robustness. For the current candidate design center, uniformly random points from an L_p ball around the point are sampled using an algorithm from [CDT98]. By assessing if these points are feasible, the algorithm can estimate the robustness of the design center. In the next step, the algorithm adapts the design center and the estimated maximal radius. Additionally, from the covariance matrix of the sampled points, the algorithm stores a linear transformation to skew or stretch the ball around a point. This way, the neighborhoods are not restricted to balls $B_r(m)$ around the point m, but rather to linear transformations of such balls, i.e. $AB_r(m)$ for a matrix A with $|\det(A)| = 1$.

Figure 5.6 shows examples of possible design centers in two-dimensional projections like those used for Figure 5.5 (without the artistic rendering). These are mappings for the audio filter application to the Odroid XU4 platform. We see how the shape of the balls is also stretched and rotated by the covariance matrix.

These methods all rely on the metric space structure of the mapping space, as distances between mappings and the concept of neighborhoods all rely on the metric space structure. Strictly speaking, neighborhoods can be defined on any topology, but not with radii as in the L_p algorithm. In [Hem+17] we used the `SimpleVector` representation for this algorithm. Here we also consider other metrics, as discussed in Section 4.2. In particular, we use the `MetricSpaceEmbedding` representation for mappings as described in 4.3. When considering other metrics, this also plays

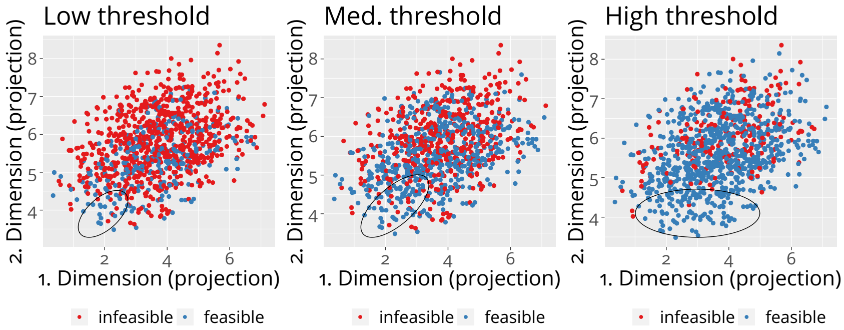


Figure 5.6: Examples of possible neighborhoods around design centers in two-dimensional random projections of the design space for the `audio filter` application on the Odroid XU4.

a role in the perturbation analysis. In [Hem+17] we define a perturbation to be a change in the mapping of exactly one process. This is equivalent to a point with distance 1 in the `SimpleVector` representation from the perturbed mapping. Equivalently, if the change is uniformly at random, selecting such a point is equivalent to a uniformly random point in the ball $\overset{\circ}{B}_1(m) = B_1(m)\setminus\{m\}$ without its center. We can generalize this to use the distance metric in the used representation to perturb the points, selecting (uniformly) random points from the ball $B_r(m)$ in the corresponding representation.

To evaluate our methods, we re-implemented the L_p adaptation algorithm and the corresponding perturbation tests in `mocasin`. From 1000 random iterations we take the $1.$, $2.$ and $3.$ quartiles of the execution time for each of the three `CPN` applications on each platform, and we use those as thresholds to have a high, medium and low feasibility threshold level, respectively. We then execute the bio-inspired design centering flow with 10 different random seeds for each of the threshold levels and compare centers to other (random) mappings for their stability using a perturbation test.

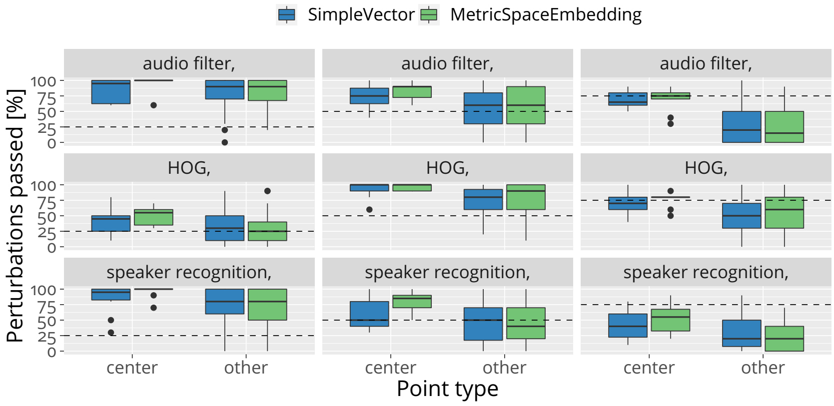


Figure 5.7: Design centering and perturbation stability for multiple threshold levels in the Odroid XU4 platform.

Figure 5.7 shows the results of this method for mapping the `CPN` applications onto the Odroid XU4 platform. We show the results for each application separately, for each of the three threshold levels. The thresholds are drawn onto the figure as horizontal dotted lines. This means that if after the perturbations mappings stay above the line, then the mapping is robust. In almost all cases the design centering strategy indeed produces robust mappings, such that they are more robust than other (randomly chosen) points in a statistically significant manner. For the Odroid XU4 platform we can see that the `MetricSpaceEmbedding` representation yields better results, albeit only slightly so. We can see how this compares to the more complex platform, MPPA3 Coolidge.

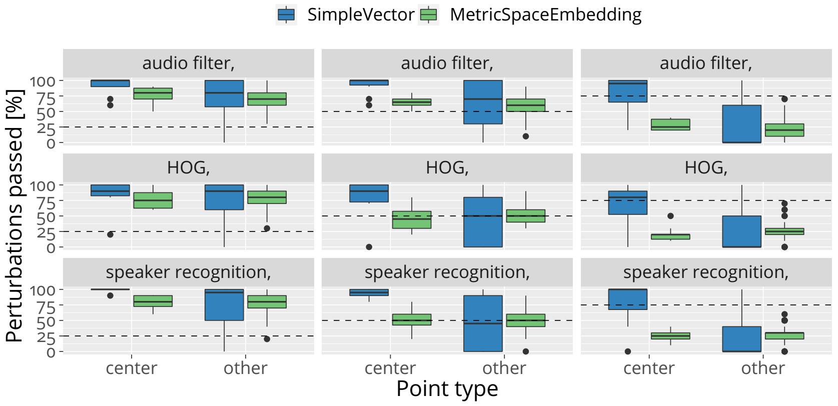


Figure 5.8: Design centering and perturbation stability for multiple threshold levels in the MPPA3 Coolidge platform.

Figure 5.8 shows the results of the design centering via L_p adaptation on the MPPA3 Coolidge platform. In contrast to the simpler Odroid XU4, the `SimpleVector` representation yields clearly better results in this case. Probably the so-called "curse of dimensionality" affects the algorithm here more, since it relies on traversing the hypervolume of the space, which is more difficult with a larger dimension. Overall the L_p algorithm seems to produce even more robust mappings in this case, yielding generally very good results. A possible explanation for this is again the size of the design space. In a larger space, with more points, it is plausible to also find larger neighborhoods of feasible points.

In this section we have successfully used metric space interpretations of the mapping space to adapt the L_p adaptation algorithm to find robust mappings. The mapping space, however, is a discrete space, while the L_p adaptation assumes a continuous design space. In the algorithm we deal with this problem by approximating a point to the closest mapping representing it. However, a better strategy might be to adapt the L_p adaptation algorithm itself to work on a discrete space. This is a promising avenue for future work in this direction.

5.3 Design Space Exploration

As we saw in Chapter 2, a central step of model-based software synthesis is a DSE step for finding mappings, among others. We know the mapping space is intractably large and complex and we cannot find the actual optima in the space for any real-life problem sizes. The best we can hope for are near-optimal mappings in a reasonable amount of time. Thus, we focus both on the quality of the mappings as well as the time required. This section will focus on DSE for finding near-optimal mappings, as defined in

Section 2.4. We will see many applications of the structures defined and analyzed in Chapter 4

5.3.1 *Heuristics and Metaheuristics*

Generally in DSE we distinguish between two approaches for dealing with these kinds of intractable problems, heuristics and meta-heuristics (cf. Section 2.4). Recall that mapping heuristics are domain-specific algorithms that exploit the specific domain-knowledge to find a solution based on a pre-defined model of the problem, whereas meta-heuristics rely on an iterative evaluation of the points. As we outlined above, different heuristics and meta-heuristics come with trade-offs between the exploration time required to find a solution and the quality of said solution. This is certainly the case for many discrete optimization problems in general, the mapping problem being no exception [Goe+16]. Commonly, meta-heuristics tend to find better results provided enough time, but require accordingly more time to do so.

A particular difficulty of comparing mapping approaches and algorithms are the different models used by different algorithms [Goe+16]. With the `mocasin` tool we designed a common framework that allows us to compare between mapping algorithms [Men+21]. In particular, in `mocasin` we have two heuristics for mapping: the GBM heuristic [Cas+12] and a static mapping variant [Men+21] of the CFS scheduler from Linux. Additionally, we have implemented genetic algorithms based on and inspired by those found in Sesame [ECP06; QP14; Goe+16], a simulated annealing [Ors+07] mapping algorithm and a tabu search [MEP08]. We also have a simple random walk algorithm for reference. A survey of these mapping algorithms, among others can be found in [Sin+13]. We implemented these algorithms for `mocasin` and this thesis to have a basis for comparison from established literature.

We first compare these mapping algorithms to establish a baseline. We execute a random walk 500 random iterations. For the genetic algorithm we run an evolutionary $\mu + \lambda$ strategy for 20 generations of population size 10, crossover rate of 1 with probability 0.35 and mutation probability 0.5, with a tournament selection with tournament size 4. For the GBM algorithm we set the parameters as `bx_m` of 1, `bx_p` of 0.95, `by_m` of 0.5,`by_p` of 0.75, The simulated annealing heuristic we execute with an initial temperature of 1 and a final temperature of 0.1, with a temperature proportionality constant of 0.5 and a random movement starting radius of 5. Finally, for the tabu search mapper we set a maximum of 30 iterations, each of size 5 and with a move set size of 10 and tabu tenure of 5, and a random candidate move update radius of 2. These parameters were not chosen systematically (e.g. using something like Bayesian optimization or general (hyper-)parameter optimization approaches), but through manual testing on examples to find sensible defaults. A deliberate choice in the parameters, however, is that the exploration times should be comparable between the meta-heuristics, i.e. such that the iterative mappers evaluate a similar amount of mappings.

Figure 5.9 shows a comparison of the different heuristics and meta-heuristics on the E3S benchmarks. Each of the metaheuristics that require random data we execute 10 times and show the variation as calculated by the unbiased estimator of the standard deviation of the multiple sampled times. The execution times vary obviously depending on the differ-

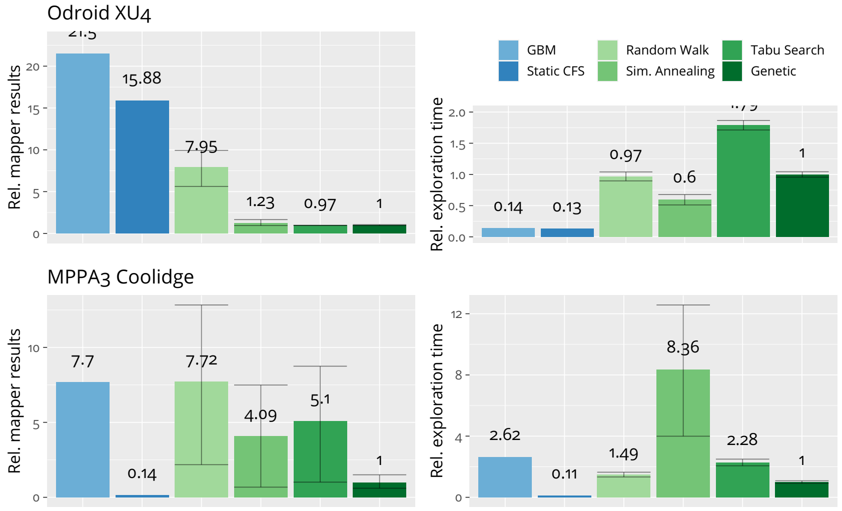


Figure 5.9: Comparison of multiple mapping heuristics and metaheuristics on the E3S benchmarks, relative to the results of the genetic algorithms.

ent benchmark applications and on the platforms they run on. The absolute values of these times, however, are not interesting for comparing the mapping algorithms. We thus norm the values of the execution times, taking the results of the genetic algorithms as baseline. We then aggregate all values with the geometric mean. The error bars in the plot are calculated by taking the average value $\pm$ the estimated standard deviation and norming each of the two extremes, the results of which are the extremes shown in the plot.

We first examine the results for the Odroid XU4 architecture. The two heuristics find comparatively worse results than meta-heuristics, in average, but they do so in considerably less time. More concretely, they yield about an order of magnitude worse results in about an order of magnitude less time. The results of the random walk heuristic are significantly worse than those of the more structured metaheuristics, even though it takes a comparable amount of time. This is the case since, as explained above, the number of random mappings was chosen specifically to be comparable to the number of mappings evaluated by the other metaheuristics. Since 500 mappings is not a small amount, it is not terribly surprising that the random mapper beats the two heuristics. Finally, the best mappings are found by the simulated annealing meta-heuristic, albeit only by 3% compared to the genetic algorithm.

When we turn our attention to the significantly larger and more complex MPPA3 Coolidge architecture, we see that the picture changes drastically. The marked difference between heuristics and meta-heuristics disappears in this case. The GBM heuristic is on par with the random walk results in average, while taking substantially less time. This is simply explained by the significantly larger design space of mapping to the MPPA3 Coolidge. In this case, the genetic algorithm significantly outperforms both other meta-heuristics, by a factor of $4-5$, while taking less time. The most striking result here, however, is the extremely good performance of the static CFS heuristic. This good performance is misleading at first. A perhaps more honest assessment of the results is that *all other (meta-) heuristics perform poorly*. We can interpret this as a consequence of the growing design space and its complexity, which affects the metaheuristics, while the static CFS mapper can still leverage domain-specific knowledge to find fairly good mappings.

5.3.2 *Leveraging Symmetries*

As motivated when discussing them in Section 4.1, symmetries can be used to improve DSE in the mapping problem. There are two distinct applications of symmetries in DSE. The first application is for speeding up meta-heuristics (without modifying them), as shown in [GSC17], by leveraging the equivalence of symmetric mappings in a symmetry-aware cache. The second application is by pruning the design space as seen by the meta-heuristic, effectively changing the meta-heuristic [GMC18; GNC].

We will first discuss the idea of a symmetry-aware cache. As discussed before, meta-heuristics work through an iterative principle, where they evaluate mappings and drive the search based on the results of the evaluation. While the evaluation is fast and light-weight by design (cf. sections 2.5 or 2.3), it still usually dominates the execution time of the exploration (cf. Figure 5.9). A defining property of the symmetries is how simulation results are invariants of the equivalence classes of orbits (cf. Section 4.1). This means that if we know the results of a simulation for a mapping, we know the results for all mappings in its equivalence class. We can leverage this by designing a symmetry-aware mapping cache, which stores simulations results by equivalence class instead of storing them for each mapping [GSC17]. This yields a trade-off, where computations about the symmetry have to be executed every time a mapping is going to be looked-up in the cache or evaluated. Ideally, these calculations would require but a fraction of the time saved on simulations.

We implemented a symmetry-aware cache in `mocasin` which uses `mpsym` and its Python interface. We used this to evaluate the method of symmetry-aware caching on the E3S benchmarks by accelerating the various meta-heuristics discussed in Section 5.3.1. We can also evaluate the domain-specific methods of mpsym [GNC] by applying this method to multiple architecture topologies. In addition to the Odroid XU4 and MPPA3 Coolidge, which we used consistently throughout this thesis, we also test the methods on the exploration of two additional architectures: HAEC and a simple generic cluster. The HAEC architecture (cf. Figure 1.2) is a PCB design with low-latency optical interconnects on layers with a regular-mesh structure (we modeled it as a 4×4 mesh). Multiple such layers (we model 4) are then connected, using low-latency wireless interconnects to communicate between adjacent layers. While in the HAEC design, each node of the layer is an MPSoC, we model the topology by placing cores in those nodes and considering the board as a single MPSoC. This serves to evaluate our methods on this topology. The generic cluster architecture we evaluate is the simplest non-trivial clustered architecture topology possible. It consists of two identical clusters, each of which with two identical cores. Each cluster shares a cache, and the two clusters can communicate over main memory.

To manage the sheer amount of experiments ($\gg 10^5$) for this evaluation and the upcoming ones in this chapter, we slightly modified the parameters of the meta-heuristics, reducing the overall execution time. We reduced the number of generations of the genetic algorithm to 10, the number of iterations of the random walk to 300, reduced the initial temperature of the simulated annealing heuristic to 0.1 and the maximum number of iterations of the tabu search algorithm to 5, with an increased radius of 2.5.

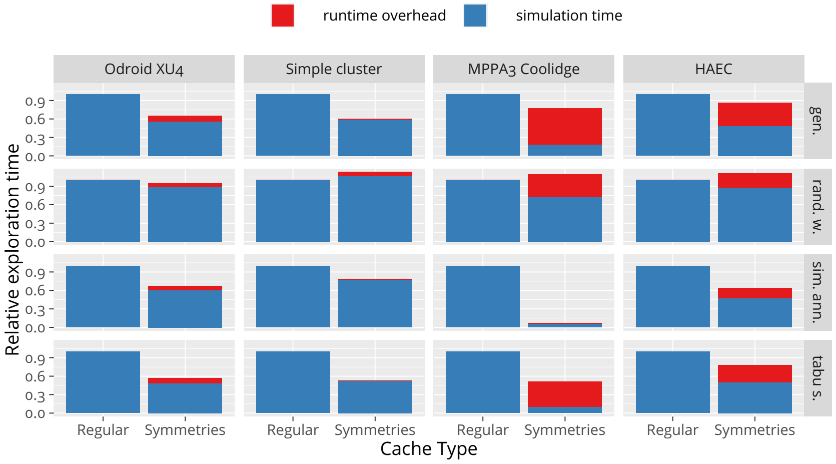


Figure 5.10: The effect of a symmetry-aware cache on multiple architecture topologies as evaluated on the E3S benchmarks.

Figure 5.10 shows the results of evaluating a symmetry-aware cache on the different meta-heuristics on different architecture topologies. It reports the execution time of the full exploration, separating the overhead from symmetry calculations from the rest of the exploration time for the symmetry-aware cache. These relative times are normed such that the exploration using a regular (non symmetry-aware) cache has a time of 1. Both variants were executed with identical seeds, for 10 different seeds and for each benchmark. Thus, they executed the exact same exploration, evaluating the exact same mappings and returning the same result.

Unsurpisingly, the symmetry-aware cache does not offset the overhead of symmetry calculations for the random walk meta-heuristic. Since the mapping space is extremely large, the probability of finding two equivalent mappings at random is quite small. Thus, both a symmetry-aware cache and a regular cache are not very useful for an unstructured random walk. The other meta-heuristics are much more structured, and clearly do benefit from the cache. For the Odroid XU4, the symmetry-aware cache consistently yields a large speedup of the exploration time, around $1.4 - 1.7\times$. For the simple cluster and the HAEC architectures, the results are similar. The best results are achieved for the MPPA3 Coolidge topology, which despite its complexity has a very well-defined structure we can exploit with our wreath-product construction (cf. Section 4.1). For the simulated annealing meta-heuristic, our symmetry-aware cache sped up the simulation in *average* by $14.5\times$!

We see that our `mpsym`-powered symmetry-aware cache is useful for speeding up DSE. It can still be improved, however. In [GSC17] we also included application symmetries in the cache, which actually made a significant difference, as we were not using the more optimized algorithms from `mpsym`. Application symmetries have much potential, which is also intuitive if we consider the cardinality of the mapping space $|V_A|^{|V_K|}$ grows exponentially with the number of processes. However, as explained in Section 4.1.1, we have no systematic method of reliably detecting these application symmetries yet, which is why we do not include them in `mocasin`.

The other improvement comes from partial permutations. While we described them in Section 4.1.4 and have a partial implementation in `mpsym`, we still need efficient algorithms on partial permutations for exploiting the partial symmetries of the mapping space. This is evident in the comparison between the results of the MPPA3 Coolidge and HAEC topologies in Figure 5.10. The HAEC topology has potentially useful partial symmetries in each of the meshes (cf. Section 4.1.4), and it also has potentially useful partial symmetries in the symmetry group of the clusters (the larger group from the wreath product). Since the first and last layers cannot communicate with each other directly (the topology is not toric), the symmetries of the clusters are all only partial.

We can also leverage symmetries in DSE by changing the underlying space as seen by the meta-heuristic. We also implemented this in `mocasin` by using the `SymmetryRepresentation` as described in Section 4.3. In this way, the meta-heuristics see the factor space and are effectively changed in their operations.

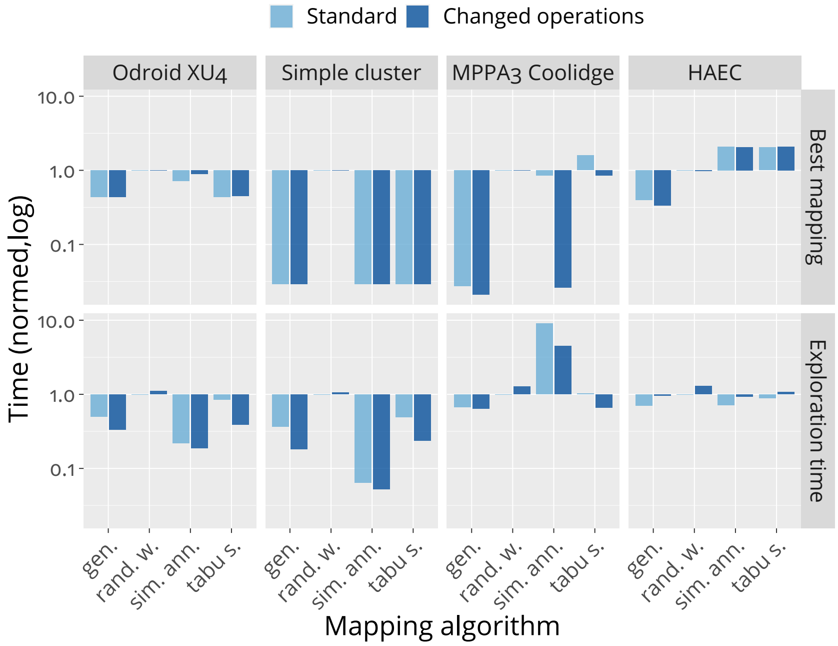


Figure 5.11: The effect of symmetry-pruning of the DSE by changing the operations in algorithms to consider symmetry. Evaluated on multiple architecture topologies on the E3S benchmarks.

We tested this with the same setup as before, the results of which can be seen in Figure 5.11. It shows both, the relative results of the mapper (in terms of the runtime of the best mapping) and the relative exploration times. The times shown are relative to the random walk meta-heuristic in the regular implementation, i.e. without symmetries. It is not surprising that most times for the best mapping results are thus < 1, as the random walk meta-heuristic is not as good as the other meta-heuristics. The more interesting comparison, however, is between both variants of each mapping algorithm. For the simpler platforms, Odroid-XU4 and the simple cluster, the symmetry-enabled variant speeds up the execution in a fashion similar to the symmetry-enabled cache, while yielding similar results. For the more complex platforms, HAEC and the Kalray MPPA3 Coolidge, the results vary between both. The results of the mapping algorithm are considerably better on the MPPA3 Coolidge, while being very similar for both representations in the HAEC platform. The exploration time was also improved consistently for both platforms. The difference between the results for the HAEC and MPPA3 Coolidge platform can again be explained by the missing partial symmetry support in the implementation.

The performance improvement for the MPPA3 Coolidge, however, is very impressive. For the simulated annealing meta-heuristic, the mappings found for the E3S benchmarks were an *average* of $32.4\times$ better with the symmetries than without them. This impressive result is, as we have seen in Figure 5.9, rather a testament of how badly the meta-heuristic performs without the symmetries for this complex architecture. To understand the mapping results for the MPPA3 Coolidge, Figure 5.12 shows the results for this platform in more detail. Instead of separating the benchmark by domain (cf. Table 3.1), we separate them by the number of tasks in the task graph. The number of tasks is a metric that better describes the complexity of the mapping space. In this case the values are normed to the regular variant of the algorithm for each algorithm, instead of norming all values relative to a single algorithm.

Figure 5.12: The effect of a pruning via symmetries on the MPPA3 Coolidge as a function of the number of tasks in the application as evaluated on the E3S benchmarks.

It is obvious that the symmetries are not advantageous for a random walk, as explained before due to the sheer size of the mapping space. This is clearly visible in Figure 5.12. For the algorithms based on tabu search and simulated annealing, the advantage comes predominantly for smaller task graphs (with $2-5$ tasks). For larger tasks probably the symmetries are not as effective anymore and both variants perform poorly. The genetic algorithm seems to continue to profit from the symmetries for larger task graphs, but it is conceivable that this advantage would still not hold for task graphs larger than those found in the E3S benchmarks.

5.3.3 *Leveraging Metric Spaces*

The same way we used the `Symmetry` representation to improve DSE, we can use the `MetricSpaceEmbedding` representation and see if this improves the performance of mapping algorithms. The two meta-heuristics that are primarily based on an underlying metric space structure of the search space are tabu search and simulated annealing. We compare the results of these two meta-heuristics on both the E3S and CPN-based benchmarks. For each meta-heuristic, each representation and each benchmark application, we measure the results of 10 runs with different random seeds.

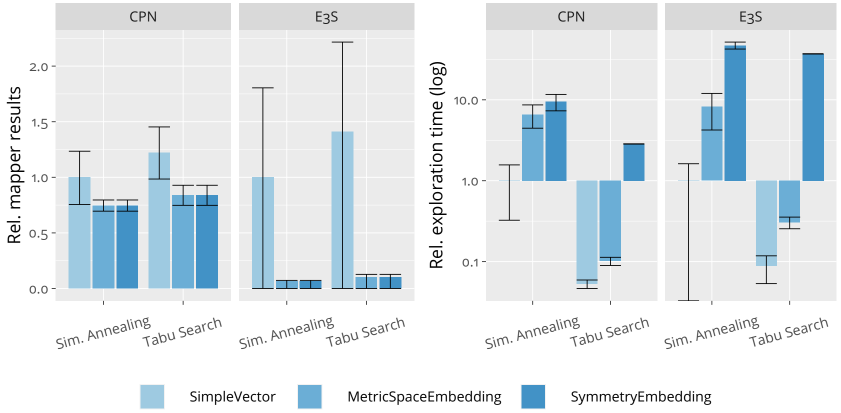


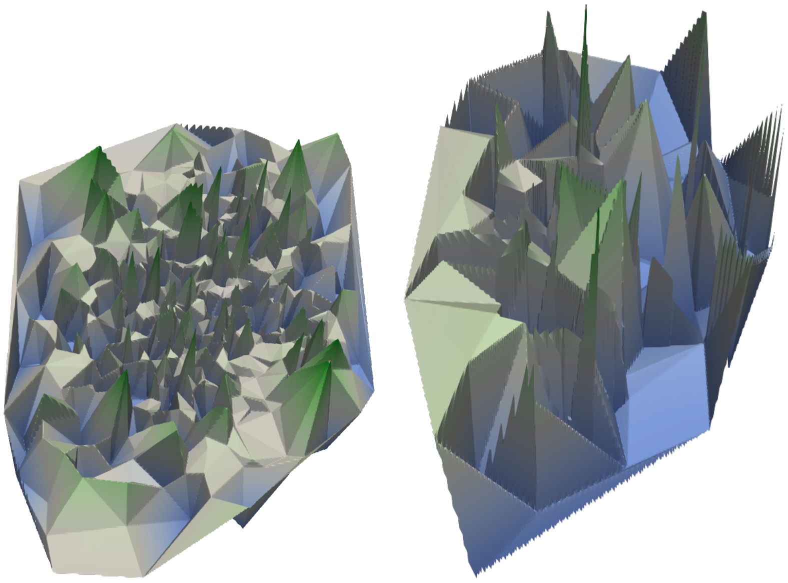
Figure 5.13: The effect of embedding-based representations in metaheuristics that leverage the geometry on the MPPA3 Coolidge platform.

Figure 5.13 shows the results of this experiments for the MPPA3 Coolidge platform, where we have seen before that the metric space structure of our embedding-based representations is better than the canonical metric in the `SimpleVector` representation. We see that the results of the exploration are significantly better for both meta-heuristics with the representations based on this better distance metric. This modest improvement does come at a cost in exploration time, which is clearly attributable to the increased dimension of the embedding spaces in these representations. We have also seen in the previous results that these meta-heuristics perform poorly for the MPPA3 Coolidge with its complex topology and large design space.

The results provide some insight into the nature of the design space and the usefulness of the embedding-based representations. We can concretely make two observations from these results and combine them to produce a new meta-heuristic. The first one is the knowledge that geometry-based heuristics indeed benefit from a better metric, independent of if the resulting heuristics are overall good. The second observation is more subtle, it's about the general structure of the design space. The design spaces of mappings seem to consist of multiple islands of performance with similar properties, separated by poorly-performing mappings. This is apparent already in the extremely simple two-task example in Figure 4.20. The visualization with the methods from [Li+18a] in Figure 4.12 or in Figure 5.5 makes this even more apparent.

The idea behind the representations is leveraging the underlying structure of the design space. As such, these "performance islands" seem to be reduced by both representation methods discussed here, as is also suggested by Figure 4.20. We also applied the methods from [Li+18a] to the same design space of the `audio filter` benchmark on the MPPA3 Coolidge with the `SymmetryEmbedding` representation. Figure 5.14 shows the results of this visualization alongside the same mapping space visualization for the `SimpleVector` representation, reinforcing the intuition of these "performance islands".

The best indicator of this, however, are the vertical lines in the relative distances from the metric spaces (cf. Figure 4.13 and Figure 4.15). Since the



(a) SimpleVector (b) SymmetryEmbedding

Figure 5.14: Visualization of the same design space of the audio filter benchmark on the MPPA3 Coolidge platform in two different representations.

abscissa in these two plots represents the (relative) mapping distance in the corresponding metric, vertical lines indicate equidistant points. The fact that there are points with the full range of relative execution times in these equidistant ranges is consistent with such "performance islands". If the mapping space landscape were more uniform, the plots would more closely resemble those where we only considered the maxima (cf. Figure 4.14 and Figure 4.16 respectively).

We can use this observation to derive a new meta-heuristic for exploring the mapping space. Our "performance islands" hypothesis implies the mapping space is full of local minima. Guiding a local search towards an optimum should thus not be as conducive to good results. Instead, we can use a simple and fast meta-heuristic to find a local minimum quickly and apply it to multiple points spread around the design space's geometry. As meta-heuristic for finding local minima we use the well-known gradient descent optimization algorithm with the momentum method [RHW86]. For the step-size we use the Barzilai-Borwein [BB88] method. In its regular form, this heuristic will quickly get stuck in a local minimum and produce poor mapping results, as confirmed by experiments (which we omit here). However, we can add a simple additional meta-heuristic to leverage the "performance islands" hypothesis. We start the heuristic at multiple random points, uniformly distributed in the design space, as defined by the distance metric. In these spread-out locations we execute (parallel) gradient descent optimizations which we cancel as soon as they reach a local minimum, which empirically happens after a handful of iterations. The meta-heuristic returns the fastest mapping found in any of the different starting locations. We configure the meta-heuristic to run on 5 different locations with a maximum of 20 iterations each, even though this maximum is almost never reached in practice in the experiments.

Figure 5.15 shows a comprehensive comparison on the Odroid XU4 platform for both benchmarks of all meta-heuristics, including our new

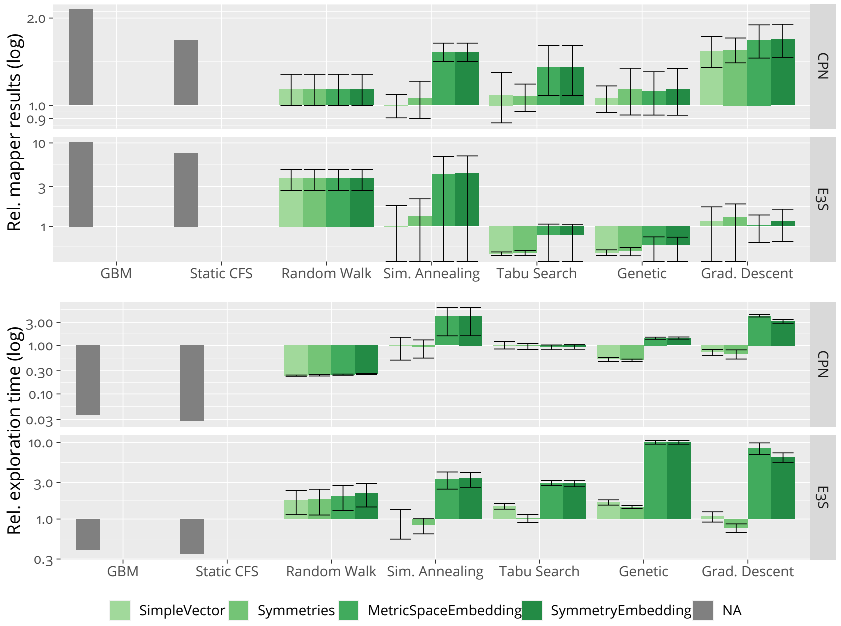


Figure 5.15: Comparison of the effects of multiple representations on the Odroid XU4 platform.

islands-based gradient descent method. The results reflect our previous results (cf. figures 5.11 and 5.9). We see that the embedding-based representations yield worse results for tabu search and simulated annealing, and require more time in almost all cases. These representations are not very useful for this platform, which is consistent with the previous results. The more interesting results are for those for the considerably more complex topology of the MPPA3 Coolidge.

Figure 5.16 shows the same comprehensive comparison of all heuristics and meta-heuristics with all representations on all benchmarks, this time for the MPPA3 Coolidge platform. Besides from the results already seen and discussed before in figures 5.11, 5.9 and 5.13, the most notable results are the results for our new island-based gradient descent meta-heuristic. Despite its simplicity, this meta-heuristic significantly outperforms all the other more sophisticated meta-heuristics. The two embedding-based representations yield worse results, which is consistent with our “performance islands” hypothesis. In a more structured design space it is more difficult to find the better performance islands than in a less structured one which features good mappings in more islands. The `Symmetries` representation yields the best results, which is also consistent with this, as it only reduces the size of the design space without fundamentally changing its topology. In a smaller space it is more likely to find an island with a higher local minimum.

Recall the intuition of Figure 4.20 of all representations for the simple two-task example. We can see how the `Symmetries` representation has two islands, with fast mappings, whereas the `SymmetryEmbedding` representation only has one. For such a simple example the symmetries capture all different execution times. In the original task graph, a GSM application from the E3S benchmarks, the differences in execution times were much lower. Figure 5.17 shows the actual mapping space. Since the differences in execution times are barely visible from the color coding, the

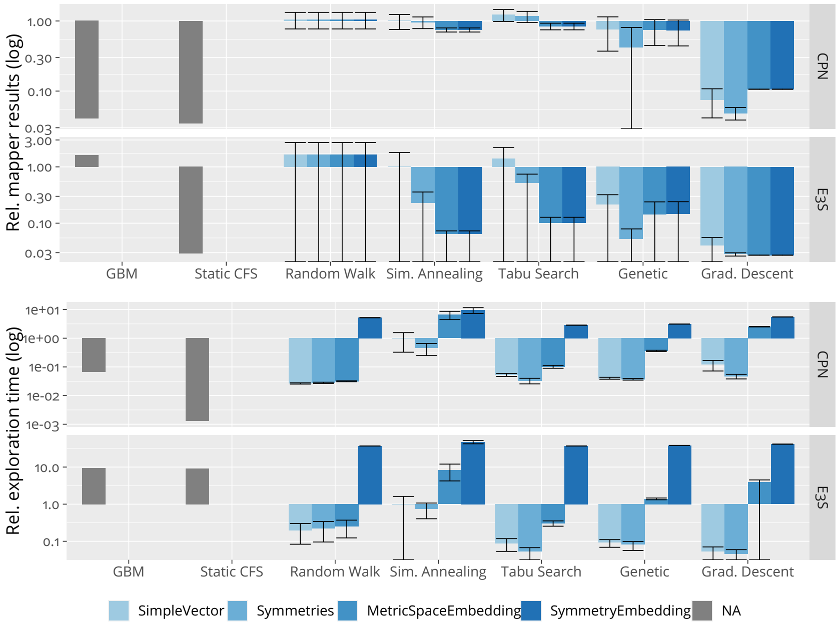


Figure 5.16: Comparison of the effects of multiple representations on the MPPA3 Coolidge platform.

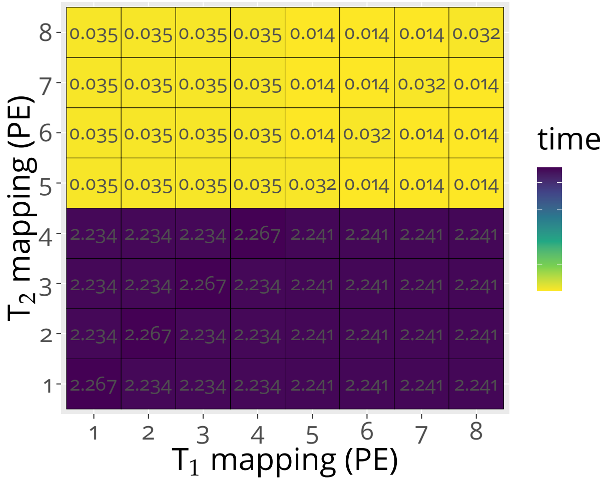
times (in ms) are also explicitly written in the figure to show the differences. This shows one of the limitations of the symmetries and a possible explanation for the “performance islands” hypothesis: many mappings are not equivalent, but very similar. In future work we could capture these similarities in mappings in a more formal fashion, e.g. using partial symmetries as motivated at the end of Section 4.1.4. This should allow us to better explore the design space.

5.4 A Vision of IoT Mappings

The Internet of Things (IoT) is a term used to describe the phenomenon that many modern embedded devices have internet connectivity and can all be interconnected this way. This opens many opportunities for automation and increased interconnectivity. Sensors and actuators in IoT devices can be combined to add new functionalities to the system that would not be possible for the individual devices. In [Web19] a case is made for *spatial ontologies* through a logical language they call *semantic localization*, to reason about distances in the real world in multiple representations, as found in the different IoT devices. These different representations of distances are not unlike the different representations of mappings discussed in Section 4.3. In analogy to semantic localization, in this section we describe how we can define a logical language that permits querying and combine representations of mappings, or *mapping ontologies*[2], in a uniform fashion. This language is based on first-order logic and also includes implicit domains and relations to define and combine statements in the different ontological representations of mappings.

---

[2] Ontologies have a well-defined meaning in logic and theoretical computer science, which we allude to, but not precisely mean here. We use the word here in its more everyday (philosophical) sense of meaning its existence or reality.



	1	2	3	4	5	6	7	8
8	0.035	0.035	0.035	0.035	0.014	0.014	0.014	0.032
7	0.035	0.035	0.035	0.035	0.014	0.014	0.032	0.014
6	0.035	0.035	0.035	0.035	0.014	0.032	0.014	0.014
5	0.035	0.035	0.035	0.035	0.032	0.014	0.014	0.014
4	2.234	2.234	2.234	2.267	2.241	2.241	2.241	2.241
3	2.234	2.234	2.267	2.234	2.241	2.241	2.241	2.241
2	2.234	2.267	2.234	2.234	2.241	2.241	2.241	2.241
1	2.267	2.234	2.234	2.234	2.241	2.241	2.241	2.241

Figure 5.17: The actual mapping space of a GSM-based two-task application from E3S on the Odroid XU4 that inspired Figure 1.4.

Depending on the context, different representations of the mapping space might offer extremely efficient ways of answering a particular type of question about a mapping. In the `SimpleVector` representation, the question "Is Task$_A$ mapped to PE$_3$" is very easy to answer, whereas answering the question "is the expected latency between Task$_B$ and Task$_C$ below $10\mu s$?" is more difficult to answer. Conversely, in the representation based on the metric space topology, `MetricSpaceEmbedding`, with a metric defined by the communication distances between hardware resources, the difficulty of these two last questions might be reversed. In order to efficiently find mappings thus, depending on the objectives, an algorithm might want to use a representation or the other, or perhaps a combination of them.

There is a distinct advantage in defining a language, as opposed to simply defining a series of programming interfaces to the different representations, and letting algorithm programmers combine them in a programmatic way using a general-purpose language, like Python. By defining a domain-specific query language, we are creating a new level of abstraction that will hopefully allow researchers to reason about the mapping problem in new ways, transcending the simple usages to combine queries that will be presented in this section.

We go over the representations as defined in Section 4.3, focusing on the kinds of questions they are well-suited to answer. We then proceed to give examples of questions that might be asked, and how these could be combined using the language. A language was implemented by Felix Teweleitt [Tew19] for `mocasin` based on these principles. We omit the details of the actual syntax and implementation of the language, as it falls outside the contribution of this thesis.

The `SimpleVector` representation is, as described throughout the previous chapters, the typical mapping representation that uses vectors of the form $m = (p_1, \dots, p_k, c_1, \dots, c_l)$ to define the mapping. It is well-suited to answer questions of the form:

- Is task A mapped to PE_1?
- Does PE_2 execute any process?

- Do tasks A and B execute on the same PE?

The `Symmetries` representation representation normalizes mappings that are equivalent to a single (canonical) mapping, while still using the vector form (cf. Section 4.1). Some examples of well-suited questions for it are:

- Is this mapping equivalent to mapping m'?
- Do tasks A and B execute on the same PE?

The `MetricSpaceEmbedding` representation uses the communication topology to define meaningful distances between PEs and by extension, between mappings (cf. Section 4.2). This representation is well-suited to answer questions like:

- Is this mapping very similar to mapping m'? (can give false positives, as seen in Section 4.2)
- Is the expected latency between tasks A and B under $10\mu s$?

The `SymmetryEmbedding` representation combines the `Symmetries` and `MetricSpaceEmbedding` representations. As such, it combines the both their strengths and weaknesses as a mapping ontology. Other representations, not necessarily based on metric spaces, could readily be added to this language. For example, we could design a hierarchy or inclusion-based distance with a way to define a PE hierarchy with refinements (PEs $\in$ clusters $\in$ chips) and similarly for hierarchical applications.

The Language

The statements in the language refer to a mapping, i.e. every mapping in the mapping space either satisfies such a statement or it does not. Thus, the questions motivated for the different representations above can be combined in a single statement, like: "Is this mapping very similar to m' (distance $\leqslant 100$) *and not* Is this mapping equivalent to m' *and* (*there exists* a PE p such that tasks A, B and C are mapped to p *or* (the expected latency between tasks A and B is small than 10 *and* the expected latency between tasks B and C is smaller than 10 *and* the expected latency between tasks A and C is smaller than $15\mu s$))."

In this language, a special solver tries to find a solution to a statement (i.e. a mapping) or a set of such solutions by evaluating the propositions in the statement in a specific order. For example, if we have a propositional statement in conjunctive normal form, we can solve the different conjuncts iteratively. Since a mapping has to satisfy each of them, the final mapping can be found by first filtering a large portion of mappings with the strongest conjunct, and iterating from there. In his work, Felix Teweleitt designed a solver for `mocasin` which utilizes a simple heuristic with precisely this principle to solve some queries [Tew19], but there is potential for much more sophisticated solving methods.

We choose to extend propositions about mappings to first-order logic so that we can have quantifiers only valid for some specific domains, like mappings, PEs, hardware communication resources, tasks (or processes or actors), communication channels. It is clear why and how these domains are the ones we can quantify over for first-order formulas describing mappings. An additional idea would be to include physical distances (over a discrete set of distances). This can be combined with different spatial ontologies in semantic localization for the Internet of Things

(IoT) [Web19]. This way, we could define IoT-mappings that have specific requirements specified in our logical mapping language.

A vision of such IoT mappings could be the following example: A smart autonomous car enters a smart parking lot. The parking lot is dark and pretty full already, and the car is low on battery, so that it needs to find a parking space with a suitable recharge station. To navigate in this dark environment, the smart car needs to offload its pedestrian recognition algorithm to a service in the parking lot, which it does by using an ontology-powered service discovery [WAL19] mechanism. Since the large concrete structure of the mapping space blocks the signal, only some very close-by servers in the smart parking lot are suitable for offloading computation with low latency and high reliability. Spatial ontologies have to be included in the mapping query to offload the high-performance pedestrian recognition in a dark environment. Furthermore, for legal reasons, the car cannot offload some decision-critical parts of the computation to an external device. This complex set of constraints on the IoT mapping can be formulated in a mixed-ontology sentence, which includes a successor of our logical mapping language with multiple representations, as well as other IoT-ontologies like semantic localization.

Clearly this vision is very far removed from today's reality, but it explains the motivation for a logical language and mapping ontologies based on the representations as discussed in this thesis.

5.5 Run-time applications: TETRiS

So far, the applications of the structures we have discussed are primarily useful at compile- or design-time. In this section we will discuss TETRiS, a hybrid mapping approach where the structure of mapping symmetries are useful at run-time.

In Section 4.1 we saw how the symmetries of the mapping problem define multiple mappings to be equivalent. We expect mappings that are equivalent to have the same runtime or energy consumption. Indeed, the simulation results are identical for equivalent mappings. When leveraging this structure for DSE, we consider only one of the multiple equivalent mappings, disregarding the rest, since they yield identical results in a simulation. The Transitive Efficient Template Run-time System (TETRiS) approach [Goe+17] leverages this property in a complementary fashion, by selecting equivalent variants at run-time according to the current system load. While this works for a single mapping, the strength of TETRiS lies in selecting from different mappings with different properties first and using the equivalent variants to find a multi-application schedule.

We say that a design point (mapping) m_1 *dominates* another m_2 if for the objective property Θ, m_1 is at least as good as m_2: $\Theta(m_1) \leqslant \Theta(m_2)$. Recall that as defined in Equation 2.1, Θ is a multi-objective function and the comparison $\Theta(m_1) \leqslant \Theta(m_2)$ is to be understood component-wise, i.e. for each objective i, $\Theta_i(m_1) \leqslant \Theta_i(m_2)$. A Pareto point is a design point (mapping), which is not dominated by any other design points. The different mappings TETRiS chooses from are, ideally, Pareto points in the space of properties we are interested in. Figure 5.18 illustrates this for the properties of energy, performance and resource utilization. Each of the green points in the property space depicted to the right of the figure is a Pareto point. It is better than every other point in at least one of the properties (performance, energy or resource utilization). Only the red point corre-

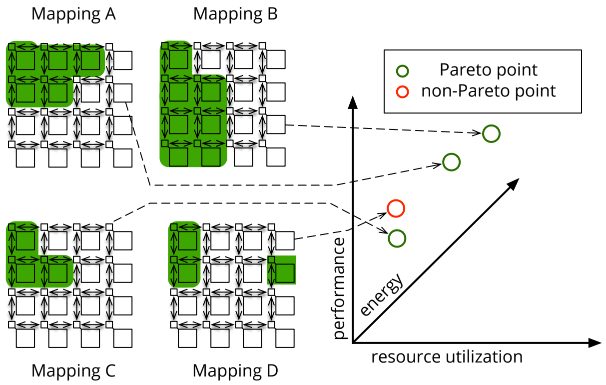


Figure 5.18: An illustration of Pareto points in the mapping space.

sponding to Mapping D is dominated by Mapping C, which utilizes the same number of resources, while being faster and more energy efficient.

The selection algorithms based on the desired properties are beyond the scope of this thesis, but `mocasin` has implementations of multiple such algorithms [KC20]. Once a mapping has been selected for each application, they need to be combined in a multi-application mapping. This is where the symmetries come into play.

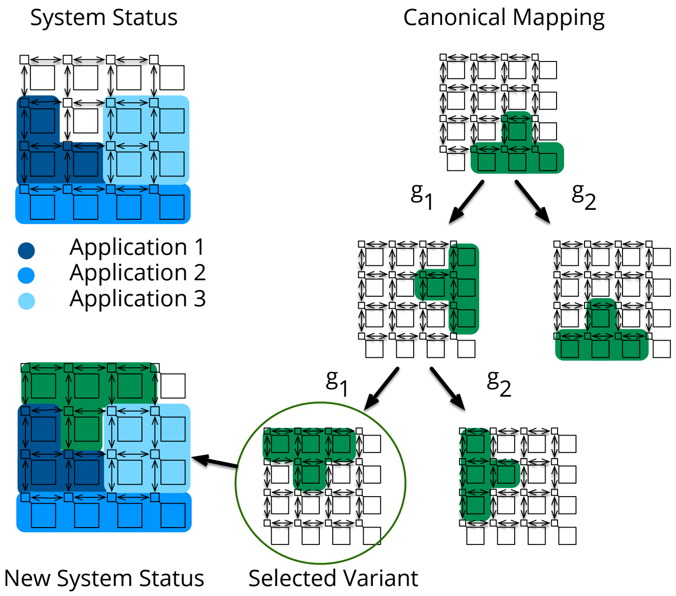


Figure 5.19: Variant selection in TETRiS.

Figure 5.19 depicts the principle behind this symmetry-enabled variant selection. At this point we assume a mapping has been selected, which we call the *canonical mapping* in reference to the canonical representative of the equivalence class (orbit). TETRiS keeps track of the system's status, knowing where other running applications are mapped. The idea behind the variant selection is then to apply the generators $g_i \in G$ of the automorphism group G for the mapping space. We call the process of applying these generators *TETRiS rotations*, informed by the geometric intuition of these transformations[3].

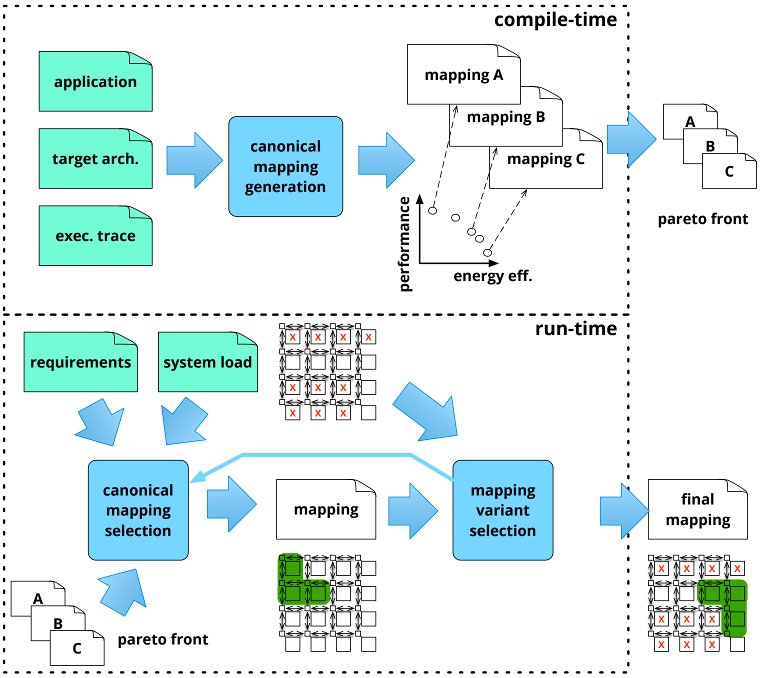


Figure 5.20: The TETRiS flow.

Combining the principles outlined above, Figure 5.20 shows the general flow of the hybrid TETRiS approach. In a compile-time phase, so-called TETRiS canonical mappings are generated as Pareto points in a multi-objective design space which ideally includes the system's resources as an objective. Then, at run-time, a selection algorithm decides which canonical mapping to use based on the requirements (e.g. real-time constraints) and possibly also based on the system's load. The selected canonical mapping is then placed onto the system by leveraging the symmetries of the mapping space in a variant selection phase, generating a final run-time mapping. When generating mappings with this method, we can guarantee the spatial isolation of computation. Provided the contention in communication is not too large, this also means that the properties of the canonical mappings are preserved (cf. Section 5.1). We tested this on an Odroid XU4 system running a pthread-based implementation[4] of the TETRiS principle [Goe+17].

Figure 5.21 shows the results of this test, running different instances of the audio filter benchmark (CPN). The three mappings T_1, T_2 and T_3 are three different Pareto points in terms of wall-clock time, CPU time and energy consumption. The mapping CFS refers to an execution using Linux' CFS scheduler and thus technically without a (static) mapping, which we measured for comparison. For each mapping we executed the four concurrent instances of the application and measured the wall-clock and execution times as well as the total energy consumption. We see that the execution of applications with TETRiS is indeed significantly more predictable than with the dynamic approach of CFS. This is especially clear for the ex-

---

[3] This might be reminiscent of the commercial game Tetris. Note that the TETRiS system is an independent research project and any resemblance is purely coincidental.

[4] `https://github.com/l3nkz/tetris`

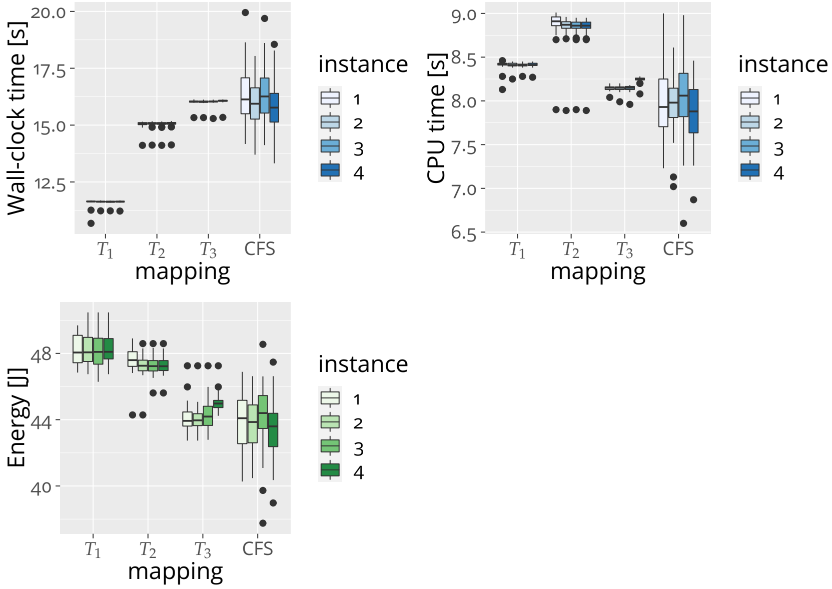


Figure 5.21: Comparison of the TETRiS system with Linux' CFS executing four instances of the audio filter benchmark simultaneously an Odroid XU4. Adapted from Figure 9 in [Goe+17].

ecution times, where the variance of CFS of around $1.3 \cdot 10^{-1}s$ is two orders of magnitude higher than that of TETRiS, which for example for T_3 is only $2.7 \cdot 10^{-3}s$. However, the difference is also very clearly visible for the energy consumption. The variance of CFS is around $2.9J$, whereas that of TETRiS for T_3 is around $6.9 \cdot 10^{-1}J$, about an order of magnitude lower.

We have already implemented the symmetries and some of the algorithms of [KC20] in `mocasin`. In future work we want to integrate the whole flow as depicted in Figure 5.20 for this. In [Men+21] we showed in preliminary results how this could yield an improvement over static mapping approaches. There we evaluated the methods on a 5G use-case. This use-case will be discussed further in Section 6.3.1. Another avenue for future work is supporting OpenMP applications.

6

BEYOND KPN: MODELS OF COMPUTATION

In his seminal paper in 1936, Alan Turing proposed what he called a "computing machine"[1]. While his machine was motivated by a person doing computations, he intended to capture the very notion of compatibility by it: namely what is possible to compute at all. He was modeling computation. Two additional such models of computation existed at the time, the λ-calculus as proposed by Alonzo Church that same year [Chu36], and the concept of general recursive functions due to Herbrand and Gödel, developed by Kleene [Kle36]. These three equivalent models [Tur37] were the original models of computation. They are equivalent in the sense that they define the same notion of what is computable. To an extent, these models were not concerned with *how* to (efficiently) compute something, but rather, *what* we can compute and what not. Since then, with the revolution of digital computers, the interest increasingly shifted to care about *how* we can compute. This spawned a much larger amount of models of computation at different levels of abstraction.

In 1972, Karp [Kar72] kick-started the field of computational complexity by identifying many problems that were equivalently difficult to compute, the class of NP-complete problems. Computational complexity relies on the fact that the asymptotic behavior of the number of steps of an algorithm, as a function of the input (size), is invariant when changing between these models of computation. Around the same time, in 1970, Dana Scott proposed a mathematical theory of computation [Sco70] based on what are now called (Scott) domains[2] and the Scott-topology. Two ideas are central in Scott's formalization. The first is a method for capturing *partial* computations, i.e. computations that have advanced but not finished yet. The second idea is that of modeling a computation as a continuous function between such domains, where a proper notion of continuity (in the Scott topology) models causality in the computation. Scott's semantics allowed to capture the process of a computation, but not the internals, which are abstracted away by the function.

The question of *how* we compute can be modeled in different ways by complexity asymptotics or partial computations in the Scott formalism, but some aspects are still left unmodeled. A significant such aspect not taken into account by these models is *where* we are computing. The theory of distributed computation was growing, with models like Petri Nets [Pet62] or seminal work like Lamport's on clocks and ordering of events [Lam78]. These models deal with properties of a computing system that has physically separate parts which split and distribute the computational load. However, the focus of the models is the system doing the computation, not the computation itself.

In this thesis we are mostly interested in concurrent models of computation. Such models abstract away the (distributed) computing system and focus on the computation itself. They consider and express concurrency in the computation, which can be exploited for parallel or asynchronous execution.

---

[1] Now known as a Turing machine

[2] Also called ω-complete partial orders [Gun92], and closely related to algebraic lattices.

6.1 An overview of Models of Computation

This section will survey some of the most important concurrent models of computation. Before diving into the models, we will first discuss the mathematical semantics[3] of computation by Scott.

6.1.1 *Partial Computation: Scott Domains*

When Scott proposed his mathematical theory of computation [Sco70], he used the term mathematical to contrast it with operational computation. In practice, the steps of a computation are defined by the ISA of the machine executing them. Most people don't write programs directly for the ISA, however. They write them in an abstract programming language, which is translated by a compiler into machine instructions. Thus, in practice, the implementation of a compiler is what informally dictates the (operational) semantics of programs. Scott's theory had the ambitious goal of being an abstraction that sat between these operational semantics and the abstract notions of computability of e.g. Church or Turing. He intended to abstract away the arbitrary implementation choices that were necessary but did not change the essence of the execution. While today his model is not the single established abstract model of semantics he sought out to define, it introduced several important ideas and mathematical structures to models of computation. In particular, a crucial abstraction introduced by his theory is that of partial computation. His theory makes it possible to express a computation as a series of partial results, without regarding the actual implementation of these. We will now introduce the basics of Scott's mathematical theory of computation.

Two related concepts can be used to computation in Scott's semantics, ω-complete partial orders [Gun92] or complete semi-lattices [LM09]. We will use the latter. Let $\langle A, \leqslant \rangle$ be a partially-ordered set (poset). For a subset $B \subseteq A$, we say a is an upper or lower bound of B if $a \geqslant b$ (resp. $\leqslant$) for all $b \in B$. Similarly, we say a is a *greatest lower bound/least upper bound* of B if a is a lower/upper bound of B and for all other lower/upper bounds a' we have $a \leqslant /b$ or $a \geqslant b$, respectively. A nonempty set $D \subseteq A$ is then called *directed* if every nonempty subset of D has an upper bound . If every such set D has a least upper bound, we say that A is directed-complete. In that case, we denote the least upper bound of D as $\sqcup D$. If A additionally has a least element $\bot \in A$ with $\bot \leqslant a$ for all $a \in A$, we say that A is a complete partial order. If, instead, A is directed-complete and every non-empty subset has a greatest lower bound, we say A is a complete semilattice.

The canonical example of this are sequences, which are a generalization of strings. Let Σ be an alphabet (a set). We call Σ^* the set of words (Kleene star) over Σ, and $\Sigma^\omega = \mathbb{N} \to \Sigma$ is the set of (countably) infinite sequences over Σ. We then define $S = \Sigma^* \cup \Sigma^\omega$ as the set of (finite or infinite) sequences over the alphabet Σ. The set of sequences S is obviously a poset with the prefix relation $\sqsubseteq$, where $s \sqsubseteq s'$ iff there exists a $t \in S$ with $s.t = s'$. Here, $(.) : S \to S$ denotes the concatenation operator (which coincidentally makes S a monoid with neutral element ϵ, the empty string). In fact, S is a complete semilattice with regard to $\sqsubseteq$ (cf. [LM09]). In Scott's model, these sequences describe the partial steps of a computation process, generating data in discrete steps (not necessarily all at once).

---

[3] Nowadays we call these semantics denotational

A function $f : S \to S$ is called monotone if for $s \sqsubseteq s'$ it holds that $f(s) \sqsubseteq f(s')$. Interpreting f as computation, this models causality: having more input data cannot change the data that has already been output. In other words, the future cannot change the past. An additional, more technical definition is that of continuity. A monotone function $f : S \to S$ is called continuous if for all directed sets D in S, it holds that $f(\sqcup D) = \sqcup f(D) := \sqcup\{f(s) \mid s \in D\}$. This concept is distinct from that of a monotone function only for infinite sequences. It means that a function will not produce its output only after reading an infinite amount of input. We call this continuous because the prefix relation defines a topology on the set S, the Scott topology.

6.1.2 *Concurrent Computation*

Scott's computation model implicitly assumed a sequential computation process, and Scott-continuous functions are a powerful method for describing partial sequential computations. Can we also use this model to describe parallel computation? Gilles Kahn did precisely this, four years after Scott published his mathematical theory of computation. He used the formalism of Scott to define a model of parallel computation, based on what he coined as process networks, now known as Kahn Process Networks (KPNs) [Kah74].

The basic idea to generalize the Scott theory of computation for concurrent execution is simple. We compose functions in networks of Scott functions, these are the KPNs. These composed functions yield a system of equations. For example, we can compose a Scott continuous function f with itself by applying it to its output. This yields an equation: $f(s) = f(f(s))$, which is solved by a fixed point of f (i.e. a sequence $s \in S$ with $f(s) = s$). A series of related results on such systems of equations and fixed-points by Tarski, Kleene and others show that such a system always has a least fixed point. This defines the semantics of KPN. For example, for the case of the single function f as above, if f is the identity function, this least fixed point is ϵ. This solves problems with loops in the system by giving well-defined semantics, and even yields a procedure to find the fixed points, by recursively applying the functions. In particular, this means that KPNs are deterministic (as per their fixed-point semantics).

There are other related models that span from the same time period, like the Hewitt-Agha actor model [HBS73; Agh86]. This was also a model of parallell computation. In it, actors communicate with other actors via messages in a non-deterministic fashion. Actors can also be dynamically created and the connections between them are also dynamic. While this yields much more flexibility, it comes with a high price: determinism.

Other models of parallel computation include Petri Nets [Pet62], in which a bipartite graph of places and transitions models the distributed execution of a system. Transitions in petri nets are very flexible as well, but they are also non-deterministic, the order in which multiple activated transitions fire is non-deterministic in general.

A series of more abstract models are the Process Calculi, which includes the well-known Π-calculus and CSP. These models are called calculi because they define specific composition rules, like parallel composition $A|B$ or $A\|B$ for processes with clear semantics. They are well-known for describing systems and specifying their behavior, e.g. in the context of model checking [BK08]. However, these are also very abstract models of computation.

Figure 6.1 shows an overview of the different models of computation and their properties. The dotted nodes refer to abstract properties of the models, whereas the filled nodes are concrete models. Concretely, the ones colored light-blue are that we review and use more in detail in this thesis. Timed models, like reactors, will be discussed in Section 6.3, and dataflow models in the section below. This figure was inspired by Figure 1.6 in [Pto14].

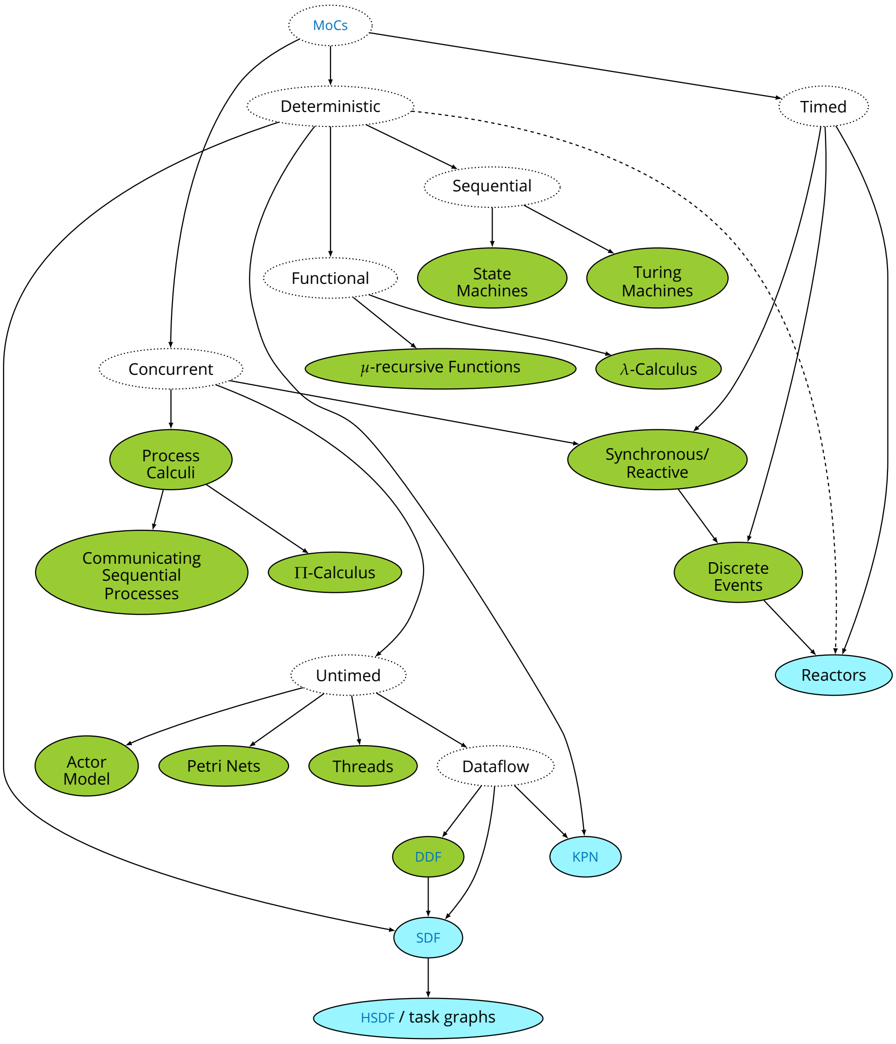


Figure 6.1: Overview of different models of computation. Color-filled nodes refer to concrete models, dotted ones are abstract properties.

6.1.3 *Dataflow Models of Computation*

A series of models stands out in the context of software synthesis and also in the domain of embedded system software, these are dataflow models of computation. More dataflow models have been proposed than what we could reasonably list and describe here. The original idea however, or at least one of the first to be published, goes back to Dennis [Den74; Den86] These dataflow models were also related with KPN, in so-called dataflow process networks [LP95; LM09]. Common among most dataflow models is the concept of actors, which encapsulate computation and which have *firing* semantics. Actors communicate exclusively via explicit input and output channels, which work as FIFO buffers. An actor fires when certain conditions are met, consuming tokens in (some of) its input channels, and producing other tokens in its output channels.

We will describe Dennis dataflow using a formalism similar to the one described in [Par95; LM09]. This formalism is very general and allows to describe many other dataflow paradigms as special cases. The basis of the formalism are the *firirng rules*. An actor has a finite set $\mathcal{R}$ of firing rules, and each rule $R \in \mathcal{R}$ is a finite tuple of words over the alphabet of values $\bar{\Sigma} := \Sigma \cup \{\bot\}$. Here, $\bot$ represents an *abscent* value, which means no data has to be present in that channel for the actor to fire. The patterns are sometimes also interpreted to be words in an extended alphabet with wildcards, e.g. $\Sigma \cup \{\bot, *\}$, where $*$ stands for any value in Σ. Note that, mathematically speaking, both $\bot$ and $*$ are unnecessary, as the empty string ϵ has the same effect as $\bot$ and $*$ can be replaced by a series of rules, one for each value in Σ. In most practical instances of dataflow, on the other hand, rules only consist of values in $\{\bot, *\}$, which is why they are very useful for descriptions.

An actor fires whenever there is enough tokens in the input channels to satisfy a rule. Here, satisfying a rule specifically means the rule R is a prefix of the channel values C, i.e. $R \sqsubseteq C$. If we include special values $\bot$ and $*$, the pattern has to be interpreted, e.g. by transforming it into the mathematically equivalent variants explained above. In this case, the tokens are consumed from the channels and the actor executes, computing something and potentially producing some outputs, which are not part of the specification in the firing rules.

Note that there is nothing preventing multiple rules to apply simultaneously. For example, an actor with two inputs could have the rules $(*, \bot)$ and $(\bot, *)$, firing as soon as one of the two channels has a token. If multiple rules apply simultaneously, there is no general order in which the actor fires and consumes the inputs. This means that this model is nondeterministic. We denote this very general, dynamic variant as Dynamic Data Flow (DDF) (alternatively, Dennis Data Flow).

If we add an additional condition, requiring that for two rules R, R' there is no upper bound S (i.e. with $R \sqsubseteq S, R' \sqsubseteq S$), then we can show that the model is deterministic. We can even relax this condition somewhat and keep determinism. In [LM09], the authors show this by explicitly constructing a Scott-continuous function from actor firings and embedding the model into KPN. They also discuss possible relaxations. This deterministic variant of (Dennis) dataflow is sometimes referred to as Dataflow Process Networks (DPN).

All these models are very expressive, so much so that they do not permit very strong analysis of their behavior. In contrast, the SDF model [LM87] has a very well-defined behavior and allows more analysis to be done statically, like scheduling or bounding the sizes of the channels [Par95]. The firing rates in the SDF model are fixed. In the formalism, this means the firing rules are always of the form $(*^{n_1}, \dots, *^{n_k})$, where $*^0 = \epsilon \mathrel{\widehat{=}} \bot$ and the n_i are called rates. Moreover, the number of tokens *produced* is also fixed statically, which is not part of the formalism of firing explained above. An apparently more strict variant of SDF is Homogeneous SDF (HSDF), in which all the rates are 1. However, these two are equivalently expressive: a well-behaved[4] SDF graph can be unrolled to an equivalent HSDF graph. The semantics of HSDF are basically equivalent with the model of *task graphs*, which are widespread in the design of embedded systems and HLS.

We discuss two additional variants of dataflow which sit semantically between SDF and DDF. The first is Cyclo-Static Data Flow (CSDF) [Bil+96], in which the static values of SDF are replaced with cycles that repeat, allowing for some controlled dynamism while retaining the analysability. Finally, Scenario-Aware Data Flow (SADF) [The+06] is a more general model which allows enabling and disabling certain paths in the graph, which are otherwise static.

Figure 6.2 shows a Venn diagram of the dataflow models discussed here and their relationship. Here we draw the distinctions as strict as possible. For example, we draw HSDF as a subset of SDF since, definitionally, it is, even though they have the same semantic expressive power. In other words, every HSDF is an SDF, and conversely, not every SDF is an HSDF, even though there exists an equivalent (unrolled) HSDF, it is just equivalent, not identical. We also include KPN and the Kahn-MacQueen (KMQ) blocking-reads semantics since they are commonly discussed as dataflow models as well. Since the models are fundamentally different, we depict them in the Venn diagram as what is embeddable semantically. Note that we depict DPN as being included in KMQ (which is proven in [LM09]), but we do not know if this inclusion is strict, in other words, if there are KMQ models which are not expressable as DPN. We will discuss the difference between KMQ and KPN in Section 6.2, where we also show that this inclusion is strict.

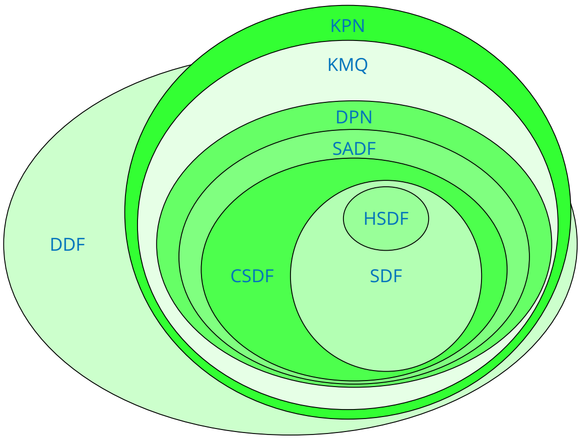


Figure 6.2: Relationships between different dataflow models of computation.

6.2 The MacQueen Gap

The KPN model was defined by Gilles Kahn in 1974 [Kah74]. While in this paper he motivated how examples of such networks could be defined, the semantics of a concrete language were only later postulated by Kahn with MacQueen in 1976 [KM76]. However, there is a gap in the semantics of formally defined networks (KPN) and the concrete networks that can be defined by the Kahn-MacQueen blocking-reads execution semantics: These concrete semantics are not as general as the formal model allows

---

[4] Concretely, a graph that can be executed without deadlocks and without an indefinite accumulation of tokens.

them to be. More concretely, there are networks which fall under the KPN formalism that cannot be expressed using the Kahn-MacQueen blocking-reads semantics. We call this gap in the semantics “the MacQueen gap”, as the gap between the formal model by Kahn and the concrete execution semantics by Kahn and MacQueen [LM09; KGC18].

In this Section we explore the MacQueen gap by showing the difference between the two formalisms, and see how we can exploit it. The contribution of this thesis is limited to the theoretical advantage from this semantics gap. The practical implementation and evaluation of the library that we describe in [KGC18], which exploits this gap in the semantics is, accordingly, beyond the scope of this thesis.

6.2.1 *The MacQueen Gap*

Recall from sections 2.1 and 6.1.2 that a KPN can be modeled as a directed graph $K = (V, E)$ where the nodes V are Scott-continuous functions $f \in V$ mapping from the set of sequences from the input channels $S_{i_1} \times \ldots \times S_{i_k}$ to the set of output channels $S_{o_1} \times \ldots \times S_{o_l}$, and the edges represent the corresponding Scoott-domains of sequences.

The Kahn-MacQueen (KMQ) blocking reads semantics are defined in a more operational fashion. The model of computation is defined implicitly by the semantics of a language [KM76], characterized mainly through blocking reads to channels. While the original semantics by Kahn [Kah74] do suggest a programming paradigm similar to the KMQ blocking-read semantics, Kahn’s original examples in a programming language made the waiting explicit in the program, not implicit in the read semantics. Neither paper aims to prove that the semantics emerging from the proposed languages correspond to the mathematical semantics of the networks in terms of Scott-continuous functions.

A central point of this distinction is the level at which these two semantics are defined: While the KPN semantics are defined at a denotational level, the KMQ blocking-read semantics are operational in nature, and thus, more fine grained. This distinction is also crucial for understanding the semantics gap, since the gap itself is operational in nature as well.

To understand the difference between the semantics we will first consider both from a denotational point of view. It is obvious that the basic semantics of the language describe a finite directed graph, and conversely, that any finite directed graph can be defined this way, by sequentially listing every node and all incoming and outgoing edges. Thus, we can think of every KMQ process as a function f mapping from the set of sequences from the input channels $S_{i_1} \times S_{i_k}$ to the set of output channels $S_{o_1} \times S_{o_l}$. The pertinent question for characterizing KMQ processes is the continuity. We sketch a proof of this in Theorem 6.2.1.

Theorem 6.2.1. A KMQ process is Scott-continuous.

Proof. (Sketch)

Let P be a KMQ process. Since P is sequential, and the reads and writes are blocking, there is exactly one sequence of read and write operations that will be executed for given inputs. This means that we can divide P into segments of execution between reads and writes, resulting in a sequence $(s_1, c_1).(s_2, c_2)\ldots.$ where for each i $s_i \in \Sigma$ is a value and c_i is the channel to/from which the value is read. We can then construct the corresponding (Scott-continuous) function f. We discuss the case for $f : S \to S$, for a single input channel r and a single output channel w, the others are analogous. Let i_1 be such that $s_1.\dots.s_{i_1}$ where $c_1 = \dots c_{i_1} = w, c_{i_1+1} = r$. The index i_1, as well as $s_1, \dots, s_{i_1}$ have to be identical for all sequences, since they cannot depend on any inputs, by defnition. We set $f(\epsilon) = s_1.\dots.s_{i_1} =: f_0$. Similarly, we let i_2, i_3 be such that

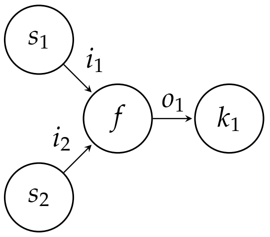
$$c_{i+1} = \dots c_{i_2} = r \neq w = c_{i_2+1} = \dots c_{i_3} \neq r = c_{i_3+1}.$$

We define $x_1 := s_{i_1+1}.\dots.s_{i_2}$ and set $f(x_1) = f_0.s_{i_2+1}.\dots.s_{i_3}$, and continue this process for all (s_i, c_i). It is clear that such a construction will produce a Scott-continuous function if it is well-defined. To see that it is well-defined we need to prove with the concrete semantics of the programming language that the same input produces the same output. ☐

Clearly, the proof sketch in Theorem 6.2.1 is not a formal proof, since we don't have formal semantics for the concrete language that defines the KMQ blocking-reads. Defining these and proving Theorem 6.2.1 properly is beyond the scope of this thesis. We get the following corollary immediately by definition:

Corollary 6.2.2. Every Kahn-MacQueen Network is a Kahn Process Network.

What about the converse implication? Can every KPN be realized by a program following the KMQ blocking-reads semantics? To understand the challenges this imposes, consider the network defined in Figure 6.3.



$$f : S_{i_1} \times S_{i_2} \to S_{o_1}$$

$$\begin{aligned} f : (x, y) &= ((x_1, \dots, x_j), (y_1, \dots, y_k)) \\ &\mapsto (x_1 + y_1, \dots, x_{\min(j,k)} + y_{\min(j,k)}) \end{aligned}$$

Figure 6.3: An example of a KPN which admits non-blocking-read semantics.

By abuse of notation, we allow $j, k = \infty$ and for $j = \infty = k$ to mean that for two streams $x : \omega \to \Sigma_{i_1}, y : \omega \to \Sigma_{i_2}$ we define $f((x_i, y_i)) = (x_i + y_i)$ for all $i \in \mathbb{N}$. The process f is, thus, a deterministic merge via addition of the two input streams and obviously Scott-continuous, i.e. a Kahn process.

Now consider the following three cases:

1. $x = \epsilon, y = (1)$
2. $x = (1), y = \epsilon$
3. $x = (1), y = (1)$

It is clear that the first two cases are prefixes of the third. By the definition of f, only this third case will generate an output (2), whereas the first two cases will result in an empty stream on the output channel o_1. However, operationally, there are different ways of processing these streams. A KMQ program has to choose to read one channel first, blocking, then read the second channel, blocking, and then output the sum. Listing 4 shows an example of code in their original language proposed by Kahn and MacQueen.

```
Process F in I1,I2 out O1 ;
 Vars x,y;
 repeat
 GET(I1) -> x;
 GET(I2) -> y;
 PUT(x+y,O1);
 forever
```

Listing 4: A deterministic merge (sum) in the POP-2-based language of KMQ.

This implementation will block in Case 1 leaving unread data in the channels, while it will execute normally in cases 2 and 3. This is because we (arbitrarily) choose to read i_1 before i_2. If we reverse this order, the implementation would block on Case 2 instead, leaving unread tokens in the channel i_1. This is relevant if we consider the execution and communication times, since e.g. there is a finite read time required to read every channel. Consider the Gantt-charts depicted in Figure 6.4. They show how blocking when reading i_1 delays the whole execution, even if i_2 could be read. This is because the blocking-read semantics forces a deterministic ordering of reading tokens when executing, whereas the KPN semantics only require the output to be deterministic, not the order of the computation itself.

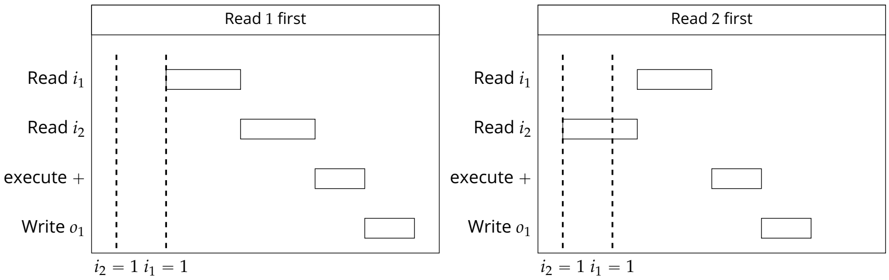
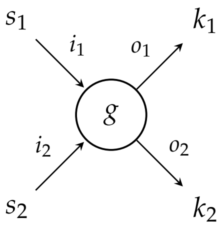


Figure 6.4: Examples of Gantt Charts corresponding to implementations of the Kahn Function f.

Having understood the nature of the semantics gap, we can thus return to the question of the other direction in Theorem 6.2.1. The gap we have shown here exposes a difference in the operational semantics, yet the different versions discussed all result in the same denotational Kahn process as defined in Figure 6.3. This does not contradict the converse direction to Theorem 6.2.1.



$$g : S_{i_1} \times S_{i_2} \rightarrow S_{o_1} \times S_{o_2}$$

$$g : (x, y) = (y, x)$$

Figure 6.5: A counterexample of the equivalence of Kahn-MacQueen and Kahn processes.

By exploiting the problem exposed in the first example, we can come up with a proper counterexample to the reverse direction of Theorem 6.2.1. The example depicted in Figure 6.5 is again clearly a Kahn process (Scott continuous), which just forwards the two incoming channels independently. In practice, this Kahn process is not very useful, but it serves formally as a simple counterexample to the equivalence of KMQ blocking-reads processes and Kahn processes. To this, consider again as inputs streams (i_1, i_2) the three cases from the first example:

1. $x = \epsilon, y = (1)$
2. $x = (1), y = \epsilon$
3. $x = (1), y = (1)$

Unlike f, the function g has a different behavior for every case:

1. $g(\epsilon, 1) = (1, \epsilon)$
2. $g(1, \epsilon) = (\epsilon, 1)$
3. $g(1, 1) = (1, 1)$

This process cannot be realized by a KMQ process with blocking reads. Assume there was such a process. Then, from the sequentiality of code, either i_1 or i_2 will be read first. Without loss of generality let us assume that i_1 is read first. Then for the input stream $(\epsilon, 1)$ however, the process will block and will never output the 1 from channel i_2, which yields the contradiction.

6.2.2 *Exploiting the Gap*

We have seen in the previous section how there is a gap in the operational blocking-read semantics proposed by Kahn and MacQueen and the denotational KPN semantics. While the counterexample from Figure 6.5 does not seem very useful, the gap in the operational nature shown in Figure 6.4 readily suggests how this gap could be exploited. In general, the Scott continuity of KPNs requires the arrival of tokens to be determistic, but it does not require the execution of independent nodes to follow the same order as the tokens, as required by the Kahn-MacQueen blocking-read semantics. Thus, as suggested by the example, the MacQueen gap can be exploited for asynchronous computation, as long as it does not break determinism.

This asynchronous execution can be used to execute multiple workers in a data-parallel fashion. Figure 6.6 shows an example of a network which does this. The worker processes $w_1, \ldots, w_n$ can exploit data parallelism by dividing a workload into different parts. This allows us to asynchronously execute the workloads, as long as we take care to preserve the order at the sink node. We can achieve this by making it part of the logic of the channels. In [KGC18] we proposed to exploit this gap and tested an implementation of this in MAPS, which modified the FIFO libraries of nodes labeled as data-parallel to relax the deterministic semantics of the KMQ blocking-reads and allowed asynchronous execution of data-parallel workers while preserving the deterministic KPN execution. The implementation of this library is beyond the contribution of this thesis, which is limited to the theoretical part of identifying the semantics gap and ways of exploiting it.

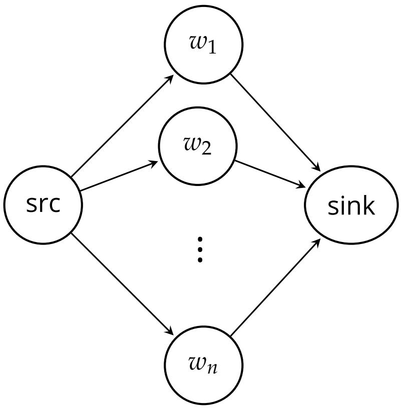


Figure 6.6: An example of data-parallelism exploiting the MacQueen gap.

6.3 Reactors

So far we have discussed multiple MoCs with different extensions. Most models we have focused on in this thesis are deterministic, which as explained in the introduction, is an important and useful property of a model's semantics. We have shown determinism in KPNs allows us to simulate and analyze their execution. Without it, many concepts we have seen in chapters 2,4 and 5 break down.

However, the models we have discussed neglect one important aspect, time. Computation takes time [Lee09], and this is a fundamental property of its semantics which is usually implicit. Determinism as we have discussed it here means that the output of a computation is a deterministic function of its input. This does not mean that the time it takes is deterministic, as we have studied in [Goe+17]. Especially in the context of CPSs or real-time systems, the computation time is an essential part of the functional specification of an application. In this section we discuss the Reactor model [Loh+19], which aims be a deterministic MoC with timed semantics.

The Reactors model is inspired by the Hewitt-Agha actor model [Agh86], which is a very widespread and well-known model of concurrent computation. The actor model is neither deterministic nor timed. Determinism in Reactors comes from combining ideas from multiple paradigms [LL19], notably, through explicit discrete-event semantics. The reactor model has two distinct time notions, *physical* and *logical time*. Physical time refers to the time as elapsing in the physical part of the system, and that part of the model is thus not part of the digital logic. Logical time, on the other hand, is the digital counterpart of physical time, and is the time that governs the computation of the reactor network. Every CPS has physical and logical time, by their very definitions. A novelty of the reactor models is making both time concepts and their separation explicit. Just as in any other timed MoC for CPSs, the two times are tightly coupled and intended to be synchronized. Making the separation explicit allows us to control the synchronization of both time models and have better control over a deterministic execution of the time logic.

Just as in the dataflow models discussed in Section 6.1, the actor model divides computation into isolated *actors* that communicate solely over explicit messages. The main difference to models like SDF or KPN is that actors and channels are not fixed. Instead, they can be dynamically created and destroyed. In Reactors, we aim to combine good ideas from multiple established MoCs. We permit dynamic re-configuration of the network through *mutations* which are well-defined (not arbitrary) transformations of the network's topology [Loh+20c]. This permits us to reason about determinism more explicitly. At the time of this writing, mutations are only defined abstractly. Specifying a set of well-defined mutations that allow us to reason about determinism and time, while still providing enough flexibility as need by the applications, is ongoing work. We will discuss this in an example use case for 5G in Section 6.3.1.

This thesis deals with model-based design in general. As such, Reactors are part of the contribution as yet another model with distinct advantages and disadvantages. Thus, apart from the design choices discussed, we only briefly outline the concepts behind reactors and a simplified denotational semantics, as well as some applications leveraging particular features of this model as opposed to other MoCs. The detailed design and implementation of Reactors as runtime systems and the corresponding polyglot coordination language, Lingua Franca[5], are outside the scope of this thesis [Loh+20a; Loh20].

Denotational Semantics

In [Loh+20c] we laid the groundwork for an operational formalization of reactors. The reactor model is a moving target and has been refined since. At the time of this writing, the most thorough and up-to-date account is in [Loh20]. Here we will deviate from the formalization both in [Loh+20c] and [Loh20], however, and attempt a denotational approach to semantics. In ongoing work with Marcus Rossel, we are using the Lean theorem prover [Mou+15] to formally verify reactors, proving properties like determinism under certain conditions. A reason for this denotational approach is that the original formalization has some mathematical inaccuracies and unspecified behavior. Clarifying or correcting these inaccuracies is necessary for having a well-defined model. The second reason for the deviation is the level of detail. We want to simplify the formalization of [Loh+20c; Loh20]. The aim of the formalization here is to isolate the abstract model's (denotational) semantics and leave implementation-specific details out as much as possible. An advantage of this formalization is that it relates KPNs and Reactors formally.

We explicitly restrict ourselves to a subset of the model, leaving out mutations and any kind of exception-handling policies. A more comprehensive (operational) model, including some of these concepts, is discussed in Chapter 2 of [Loh20]. These restrictions are in part for simplicity, but also due to this being ongoing work. At the time of this writing, we have not finished the Lean-based formalization to include these aspects. Extending a simple model is easier than changing a complete model that is problematic. It is important to note that as ongoing work, this alternative formulation has not yet undergone peer-review (as opposed to [Loh+20c]) and is subject to change.

Timeless model

Reactors are a timed model, with specific semantics of how the time progresses and what can happen when. The logical (functional) semantics of a reactor network are complex as well, however. We first begin defining

---

[5] `https://github.com/icyphy/lingua-franca`

the computational semantics of the network in a timeless fashion, and then extend the model to include time.

Computation is essentially manipulation of data. Models are thus built and defined by how they manipulate data. We follow a model of computation based on Scott's semantics of computation (cf. Section 6.1). Data is modeled as sequences $s \in S = \Sigma^* \cup \Sigma^\omega$ over a finite alphabet Σ, which we require to include a special symbol $\bot \overset{!}{\in} \Sigma$ that represents absence of data [6]. The basic unit of computation in Reactors are reactions, which take a finite number of data tokens as input and return a finite number as output. Thus, to define a reaction we simply consider a Scott-continuous function $n : S^k \to S^m$. Sequences can have different lengths (both finite or infinite), yet reactor networks execute in discrete ticks, which result in sequences of the same length. To model this we define "padding" using the special symbol $\bot \in \Sigma$ on a finite sequence $s \in \Sigma^* = S \backslash \Sigma^\omega$, by defining $\hat{s} = s.(\bot^\omega)$. Finally, an important restriction is that we want to ensure reactions in the timeless model do not to take multiple inputs from the same channel before producing an output.

Definition 6.3.1 (Reaction). Let $n : S^k \to S^m$ be a Scott continuous function. We call n a *reaction* if for any two $s, s' \in S$ such that s' is a proper prefix[7] of s, i.e. $s' \sqsubsetneq s$, then this also holds for the images under n, i.e. $n(s') \sqsubsetneq n(s)$.

Note that our definition is a restriction on the definition of reactions. As defined in [Loh+20c] with an informal *source code* "object", they can be interpreted in the semantics of the language of that source code. As such, they could implement any relation on $S^k \times S^m$. In particular, we assume reactions are deterministic (as mathematical functions) and respect causality (being Scott-continuous). Note that this does not mean they are stateless. State is implicit in the definition of a function on the complete history of inputs, as opposed to a function on a single input token. In the latter, for $f : \Sigma^k \to \Sigma^m$, state can be formalized as a "self-edge" i.e. an i, j such that

$$f(*, \underbrace{s}_{i}, *) = (*, \underbrace{s'}_{j}, *) \text{ with } s, s' \in \Sigma,$$

the $*$ being other values we don't care about here. However, we use the denotational formalization of computation of functions by Scott as complete sequences of inputs and outputs, which makes the state implicit.

Definition 6.3.1 also has strong theoretical consequences. It implies that every monotone reaction is Scott-continuous, since it is equivalent with $|f(s)| \geqslant |s|$ for all $s \in S$, which avoids the pathological cases that distinguish monotone and Scott-continuous functions. Proving this fact is beyond the scope of this thesis.

Modeling reactions as Scott-continuous functions, we do not specify anything about the *length* of the sequences. A reaction might produce a longer output sequence than its input sequence. At this stage this is not important, as we model the complete computation with a single function. We will come back to this later, in the timed model, when we relate these sequences to concrete times.

---

[6] Note that the exclamation mark in the notation before refers to the requirement of that inclusion (as opposed to a statement of a fact).

[7] Not to be confused with $s' \not\sqsubseteq s$, the negation of $s' \sqsubseteq s$.

If a reaction $f : S^k \to S^m$ has the property that

$$f(s_1, \ldots, s_{i-1}, \bot, s_{i+1}, \ldots, s_k) \sqsubseteq (\bot^\omega, \ldots, \bot^\omega) \text{ for all } s_j \in S, 1 \leqslant j \leqslant k, j \neq i,$$

we say that f has a trigger on the input i. Recall that the symbol $\bot$ represents the absence of values. Intuitively, thus, a reaction that triggers on i will not execute if there is no input on i. In other words, the values in i trigger the reaction, hence the name. Besides triggers, the original definition also has other components as part of reactions, namely sources and effects (or dependencies and anti-dependencies), scheduleable actions and a deadline. We include most of these concepts in other definitions, e.g. the reactor or the network.

To communicate between reactors (or perhaps more precisely, between reactions), we need to send and receive data. We do this using input and output ports, which we model simply as identifiers in an index or identifier set I. A reactor has a series of reactions with input and output ports, and reactors connect to each other through them.

Definition 6.3.2 (Reactor)**.** Let I be an index set. A reactor is a tuple $r = (N, D, D^\vee)$ where N is a finite poset of reactions $n : S^{k_n} \to S^{m_n}$ and

$$\begin{aligned} D &: N \to (\{1, \ldots, k_n\} \to I), \\ D^\vee &: N \to (\{1, \ldots, m_n\} \to I), \end{aligned}$$

are called the sources and effects respectively. We define the set of input ports as $\mathrm{Input}(r) = \bigcup_{n \in N} \mathrm{im}(D(n))$ and, similarly, the set of output ports we define as $\mathrm{Output}(r) = \bigcup_{n \in N} \mathrm{im}(D^\vee(n))$. We require that $\mathrm{Input}(r) \cap \mathrm{Output}(r) = \varnothing$ as part of the definition of a reactor.

The sources D make a correspondence between the indices in the tuple of input streams of a reaction and the (port) identifiers I. For example, if a reaction $n : S^2 \to S$ takes two inputs, $D(n) : 1 \mapsto c, 2 \mapsto b$ means that the ports c and b are the two input ports of n, in that order. The effects $D^\vee$ are analogous but for the outputs of the reaction.

We require N to be a poset for two reasons. Firstly, we want to be able to specify an order in which reactions are always executed. However, we also want to allow explicitly making the model non-deterministic by making reactions incomparable. When two reactions are incomparable, they are executed in a non-deterministic order. By the order-extension principle, it is always possible to execute reactions while respecting the partial order.

More formally, let $n, n' : S \to S$ be two reactions. For simplicity, we assume they have a single (shared) input and output port: $(D(n))(1) = (D(n'))(1)$ and similarly $(D^\vee(n))(1) = (D^\vee(n'))(1)$. Recall that $\hat{s} = s.(\bot)^\omega$ for $s \in S \backslash \Sigma^\omega$ is a "padding" of a sequence with absent values. We say that a function $f : S \to S$ is a *priority-preserving execution* if for all $s \in S$ and for all $i \in \mathbb{N}$, it holds that:

$$\begin{aligned} (\hat{f}(\hat{s}))_i &= (\hat{n}(\hat{s}))_i, && \text{if } n \leqslant n', (\hat{n}(\hat{s}))_i \neq \bot && (6.1) \\ (\hat{f}(\hat{s}))_i &= (\hat{n}'(\hat{s}))_i, && \text{if } (\hat{n}(\hat{s}))_i = \bot && (6.2) \\ (\hat{f}(\hat{s}))_i &= (\hat{n}'(\hat{s}))_i, && \text{if } n' \leqslant n, (\hat{n}(\hat{s}))_i \neq \bot && (6.3) \\ (\hat{f}(\hat{s}))_i &= (\hat{n}(\hat{s}))_i, && \text{if } (\hat{n}'(\hat{s}))_i = \bot && (6.4) \\ (\hat{f}(\hat{s}))_i &\in \{(\hat{x}(\hat{s}))_i \mid x \in \{n, n'\}\} && \text{otherwise} && (6.5) \end{aligned}$$

In this case we write $f \in \bigsqcup_{D, D^\vee} \{n, n'\}$. Equations 6.1 and 6.3 formalize the reaction priority when the two reactions are ordered, and Equation 6.5 the non-deterministic ordering when n and n' are incomparable.

If a reaction returns an absent value $\bot$, then the value of the other reaction is written on the output sequence. Note when $n \leqslant n'$ or $n' \leqslant n$ (which trivially includes the case $n = n'$), all functions in $\bigsqcup_{D,D^\vee}\{n, n'\}$ are equivalent up to padding with $\bot$.

This definition can be trivially generalized to more than one (shared) input or output sequence (component-wise), and to non-shared input or output sequences by requiring equations 6.1-6.5 to hold quantified over all non-shared sequences. Finally, for a poset N of reactions, we define $\bigsqcup_{D,D^\vee} N$ analogously (component-wise), requiring equations 6.1-6.5 to hold pairwise for any two $n, n' \in N$.

Reactors are connected in networks. We model these networks explicitly, separate from reactors themselves. In the original definition, this is avoided by building reactors hierarchically. There is no semantic distinction between a hierarchical model and a flat model, where all contained reactors are "inlined" in a network[8]. We prefer separating the reactors and their networks, since the definition of reactor networks allows us to specify the semantics of how they can be connected. Here, we distinguish between two cases: an untimed one, which we call timeless and represents the purely logical execution of the network, and a timed one, which is the general case and is built on top of the former.

Definition 6.3.3 (Timeless reactor network)**.** A timeless reactor network is a multigraph $\mathcal{R} = (V, E, \xi)$ with a set of reactors as nodes V, a set of edges E, which we require to be pairs of indices, $E \subseteq I \times I$ and $\xi : E \to \{\{r_1, r_2\} \mid r_1, r_2 \in V\}$. For this multigraph we require that for any two distinct reactors $r_1 \neq r_2 \in V$ the input and output ports are pairwise disjoint, i.e. $\mathrm{Input}(r_i) \cap \mathrm{Output}(r_j) = \varnothing$ for all $i, j \in \{1, 2\}$ and every edge is a tuple consisting of an output port and an input port, i.e. for all $(i, j) = e \in E \subseteq I \times I$ there exist $r_1, r_2 \in V$ such that $i \in \mathrm{Output}(r_1)$ and $j = \mathrm{Input}(r_2)$. We additionally require that the multigraph has no self-edges, i.e. $|\xi(e)| > 1$ for all $e \in E$.

Recall that a multigraph is a graph that can have multiple edges, and the function $\xi : E \to \{\{r_1, r_2\} \mid r_1, r_2 \in V\}$ defines which vertices are connected by each edge. Here, the edges themselves carry semantics as well. They define which ports specifically they connect in the reactor. We define the set $I(\mathcal{R}) = \bigcup_{r \in V}(\mathrm{Input}(r) \cup \mathrm{Output}(r))$ as the set of ports of $\mathcal{R}$.

We make an additional remark about Definition 6.3.3, namely that we don't require all ports to be connected. Indeed, some ports we explicitly want to leave disconnected to define the general, timed model.

Timed Networks

We are finally ready to introduce time into the model. Reactors are based on a logical time model of discrete events. We formalize logical time as a totally ordered set of discrete timestamps, which is order-isomorphic to the naturals $\mathbb{N}$ (or a finite subset). When two events happen at the same time, we want to keep the total-order property to distinguish them. For this, we use *superdense time* [MMP91; Pto14] , which adds *microsteps* at every time unit. Thus, time tags $t \in \mathbb{N} \times \mathbb{N}$ are lexicographically ordered tuples of natural numbers, where the first number represent the timestamp as ticks (in some specific unit of time), and the second number represents microsteps. Physical time, on the other hand, we define

---

[8] Note that this might change if we extend the model to include mutations.

as real numbers $\mathbb{R}$ to allow continuous-time physical models (e.g. Newtonian mechanics). However, computation only can interact with physical time at discrete time intervals. We compose these two types of time in a unique time object, a *tag*.

Definition 6.3.4 (tag)**.** A (time) tag $t \in \mathbb{T}$ is a value in the sum (type) $\mathbb{T} := (\mathbb{N} \times \mathbb{N}) \oplus \mathbb{R} = (\mathbb{N} \times \mathbb{N}) \dot{\cup} \mathbb{R}$, which is commonly also called the disjoint sum[9]. The embedding for the first component $\mathbb{N} \times \mathbb{N} \hookrightarrow \mathbb{T}$ is called *logical* time, and the embedding from the second component $\mathbb{R} \hookrightarrow \mathbb{T}$ is called physical time. We say that t is a *logical* or *physical* time tag respectively.

Note that Definition 6.3.4 differs from [Loh+20c; Loh20]. The rationale for this is that this definition gives us a uniform way of referring to time while still distinguishing between logical and physical time. We could also have defined $\mathbb{T} = (\mathbb{N} \times \mathbb{N}) \oplus \mathbb{N}$ taking into account only the discrete measurements of time that are available to the digital component of the CPS. This definition with the real numbers $\mathbb{R}$ instead allows the model to be combined with continuous-time models of physical time, and it adds no restrictions to our semantics.

Reactions are, in a sense, controlled functions we compute from incoming data. Some data we have no control over, like incoming input (e.g. from a sensor), or an asynchronous computation we scheduled. To model these we use *actions*. Note that these actions are more a model of (tagged) data, as opposed to reactions which are a model of computation. This creates a false dichotomy, since actions are fundamentally different from reactions. Actions are more closely related to the input and output ports, and the naming confusion might be thus easier to resolve when thinking that, in this way, reactions react to actions.

Actions are central to the model, since they are the mapping between the functional world of reactions and the time semantics. Definition 6.3.5 ensures actions do not mix the two different time types, and respect causality (i.e. an action cannot change the past).

Definition 6.3.5. Let $\mathbb{T}_{\text{discrete}} := \{| \; T \subseteq \mathbb{T} \mid T \text{ is discrete}\}$ be the set of discrete subsets of $\mathbb{T}$. An action is a partial[10] function $A : \mathbb{T}_{\text{discrete}} \to S$ such that

- For all $T \in \operatorname{dom}(A)$, the discrete set of tags T and the corresponding sequence $A(T)$ are order-isomorphic (in particular, $|T| = |A(T)|$).
- All $T \in \operatorname{dom}(A)$ are either sets of logical or physical tags, i.e. $T \subseteq \mathbb{N} \times \mathbb{N}$ or $T \subset \mathbb{R}$. We call A a *logical* or *physical* action, respectively.

For a discrete set of times $T \in \mathbb{T}_{\text{discrete}}$, with $\mathbb{T}_{\text{discrete}}$ as in Definition 6.3.5, we call $d(T) = \inf_{t < t' \in \operatorname{dom}(A)} t' - t$ the minimum delay or spacing of the time set T. Here, subtraction is to be understood component-wise, and only up to 0, as it is sometimes [Run89] defined on the set of natural numbers i.e. $(t_1, t_2) - (t'_1, t'_2) := (\max(t_1 - t'_1, 0), \max(t_2 - t'_2, 0))$. For an action A we define $d(A) = \inf_{T \in \operatorname{dom}(A)} d(T)$. Note that the formulation in [Loh20] distinguishes between the minimum delay as specified by the programmer and the minimum (time) spacing as acceptable for the runtime system. We consolidate both here, since, for simplicity, we disregard policies for when this spacing is violated and error handling in general. Consequently, we also do not model the spacing violation policy included

---

[9] In the language of set theory, that we use by convention in this thesis.

[10] See Section 4.1.4 for the required definitions.

in [Loh20]. We define it also for the time set and not the action, since our semantics are denotational and not operational.

Definition 6.3.5 allows us to associate any given subset of times to a different sequence of values. In particular, this allows us to model the timestamps themselves being part of the value in the sequence, e.g. for a reaction that stores the current time to a log file.

The order-preserving bijection between a discrete set of tags and sequences ensures a causal execution. Since the mapping is order-preserving, a going forward in timestamps can only increase the port's history (the sequence of values). Similarly, adding tokens to the port's history can only move forward in time. Moreover, since it is a bijection, it means that adding tokens to the port's history *has* to move forward in time, and vice-versa. One step in the discrete set of tags corresponds to exactly one value in the history. In particular, time has to advance every time reactions are executed. This is why we need microstep delays in logical time, so that we can execute events with identical logical timestamps. For physical time we cannot have two events with identical timestamps, but the timestamps can be arbitrarily close to each other, so this is not a very strong restriction. Note that in [Loh20] physical time gets converted to logical time when assigned a tag. In that formalism it is thus possible for two physical actions to have values with identical tags, but the tags would ultimately have different microstep units when executed, which is an unavoidable source of non-deterimnism. This is not different from e.g. adding a small enough $\epsilon > 0$ in this model.

Definition 6.3.6 (Reactor network)**.** A reactor network is a tuple $(\mathcal{R}, \tau)$, where $\mathcal{R} = (V, E, r)$ is a timeless reactor network and $\tau : I(\mathcal{R}) \to \mathcal{A}$ is a partial function of the identifier set (of ports) of $\mathcal{R}$ to a set of actions $\mathcal{A}$, such that for every $i \in I = I(\mathcal{R})$, exactly one of the following is true: $i \in \mathrm{dom}(\tau)$ or there exist an edge e in $E_{\mathcal{R}}$, such that $e = (i, j)$ or $e = (j, i)$ for an $i \neq j \in I$.

We call $\mathrm{im}(\tau) \subseteq \mathcal{A}$ the set of actions of the reactor network $(\mathcal{R}, \tau)$. Here, the mapping τ relates actions with all dangling ports in the timeless network. The last condition on τ ensures that no ports are left dangling in the (timed) reactor network.

Both our original description in [Loh+20c] and the updated one in [Loh20] are very explicit about reaction and event queues, scheduling and mutexes. These are very important aspects for any implementation of the model, yet they conflate the implementation and the semantics. Here we are interested mostly in the general concepts behind reactors, the implementation is outside the scope of this thesis. As a consequence, we rather err on the side of abstraction, by preferring to abstract away details and clarify them in future work if necessary.

Definition 6.3.7 (Execution of reactor networks)**.** Let $T \subseteq \mathbb{T}$ be a discrete set of time tags and let $(\mathcal{R}, \tau)$ be a reactor network. We denote by $\mathcal{R}_\tau(T)$ the network obtained by substituting in the timeless network $\mathcal{R}$ for each port $i \in \mathrm{dom}(\tau)$ the sequence $(\tau(i))(T)$ (recall that $\tau(i)$ is an action). An execution of $\mathcal{R}$ with discrete set of time tags T is a sequence s_i for every port $i \in I$ such that:

1. For every reactor $r = (N, D, D^\vee)$ in $V_{\mathcal{R}}$ there exists a function $f \in \bigsqcup_{D,D^\vee} N$ with $f(s_{i_1}, \dots, s_{i_k}) = (s_{j_1}, \dots, s_{j_m})$ where $i_1, \dots, i_k, j_1, \dots, j_m$ are the corresponding ports of r according to the edges E of $\mathcal{R}_\tau(T)$.

2. For every f as in 1, the sequences $(s_i)_{i \in I(\mathcal{R})}$ are a fix-point of the set of equations defined by the network $\mathcal{R}_\tau(T)$.
3. For every time value $t \in T$, at least one action A is non-absent, i.e. $|\{A \in \operatorname{im}(\tau) \mid A(t) \neq \bot\}| \geqslant 1$.

The execution of a reactor network in this way is not modeled as an iterative process. The computation itself is modeled through the sequences S in the Scott semantics of computation. The time values of actions A are chosen (non-deterministically) for an execution, modeling the non-determinism from the environment. Condition 1 in Definition 6.3.7 defines the execution priority of reactions. This, together with Condition 2 ensure that reactions have well-defined semantics. Finally, Condition 3 ensures that only one action is scheduled at a time.

A central idea behind Reactors is to split logical and physical time explicitly. However, these two time concepts are conceptually linked, since logical time is just a digital estimation of physical time. Thus, the reactor runtime should strive to synchronize these two time concepts whenever possible. This is realized by the requirement of executing events in timestamps order, ensuring logical time never goes past physical time. Nothing guarantees that the converse does not happen, however. Physical time could go far beyond logical time. In an implementation, and indeed in the formalization of [Loh20], a deadline in the reactions controls how far away logical time can lag behind physical time.

Note that the definition of reactor networks does not exclude any loops. The fix-point-based definition allow us to have well-defined semantics with such loops (cf. [Kah74; LM09]), as ensured by Condition 2 in Definition 6.3.7. In some cases, however, the least fix-point of the network might result in an empty sequence. This can be the case when the ordering in reaction causes a so-called causality loop. See Section 2.6 of [Loh20] for a more thorough discussion. Also note that Condition 2 does not require the fix-point to be minimal, but this is given by the order-isomorphism condition on actions.

In [Loh20], reactors are explicitly required to have two special actions, a startup and a shutdown action. We do not require these two actions explicitly. An empty reactor network, that does nothing, is also a well-defined reactor network, albeit a pretty useless one.

Conjecture 6.3.8 (Reactors are deterministic)**.** Let $(\mathcal{R}, \tau)$ be a reactor network such that for every reactor $r = (N, D, D^\vee)$ the set N is totally ordered. Then for every execution $\mathcal{R}_\tau(T)$ with a discrete time set T the values of s_i for every port $i \in I$ are uniquely specified.

Our formalization has allowed us to specify determinism in reactors in a mathematically precise fashion. We believe in future work we can prove Conjecture 6.3.8 by using fix-point theorems, in a fashion similar to [Kah74].

This also gives us the language to discuss different kinds of determinism: can the values be independent of the set of time tags T? In general, it cannot work. Consider a network which prints the timestamps it sees, this will never be independent of the timestamps. On the other hand, if every discrete sequence of timestamps is mapped to the same sequence of values, i.e. $A(T) = A(T')$ for all actions A and (valid) sequences of timestamps T, T', then the behavior is trivially time-deterministic. Note that in this case, Definition 6.3.5 implies that $T \cong T'$ are order isomorphic and, consequently, $T = T'$, which is a very strict condition on the network, that has to have a constant number of actions. There are certainly relaxations of this that allow us to define reasonable conditions for time-determinism.

We can even go further and distinguish between logical and physical actions for determinism. For example, we can define a reactor network to be time-deterministic if it only depends on the image sequences of *physical* actions A. The non-determinism from the physical world is outside our control, but with this definition we are also ensuring logical actions to behave deterministically as a function of the physical ones.

A final word on distributed execution, which we have ignored so far, is due here. Our semantics are denotational, they are meant to describe what is computed, not how. In particular, a distributed execution should adhere to these semantics just as a sequential one. A fundamental problem with using our model for distributed execution, however, are time tags, which are uniform in the model. Strictly speaking, we could replace our Newtonian model of time with a relativistic one and consider different frames of reference and transformations. The model, as is, can thus be considered a model for a fixed (inertial) frame of reference.

In future work we plan to consider distributed execution and its consequences (or lack thereof) on our denotational semantics. We also plan to precisely identify conditions for these different possible definitions of determinism and verify them, using the Lean theorem prover [Mou+15] and a formalization similar to the one described here. As mentioned above, this is ongoing (unpublished) work with Marcus Rossel.

6.3.1 *Applications in 5G*

Having defined Reactors formally, we consider some applications for the model. In this section we will discuss Reactors in the 5G standard.

Telecommunication standards evolve constantly, pushing the limits of signal processing systems from almost every angle. Consumer demands adapt to increases in capabilities. This results in a feedback loop that not only raises the demands themselves, but also their heterogeneity. In LTE today we already see very dynamic demands, with different users requiring very different bandwidths at different times. With the increased capabilities of 5G, the dynamicity of the demand will only increase.

Signal processing systems, however, are not built for dynamic workloads; they must tolerate the worst case. This makes sense, since a system that is capable of processing the highest demands can also process lower demands. However, parameters like user count, resource blocks supported, used MIMO scheme and carrier aggregation have a nuanced relationship in terms of resources pressure. Additionally, the sub-carrier spacing is also flexible in 5G systems. As a direct consequence, the real-time requirements have to adapt to the changing transmission time interval. All of this yields a parameter space with a large dynamic range of possible workloads.

Figure 6.7 shows a simplified overview of the uplink modem in a basestation for 5G. We see that the overview already resembles MoCs like dataflow or Reactors. Details on the requirements, like the sizes and numbers of FFT nodes depend significantly on the workload being processed. However, the dependencies between the resources required for the baseband processing and the parameters of the workload are non-trivial. Figure 6.8 shows a small selection of parameter combinations for LTE base-

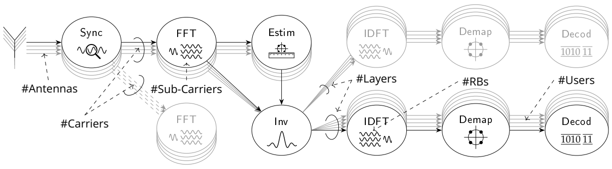


Figure 6.7: Simplified model of a basestation uplink modem. Adapted from Figure 2 of [Wit+20]

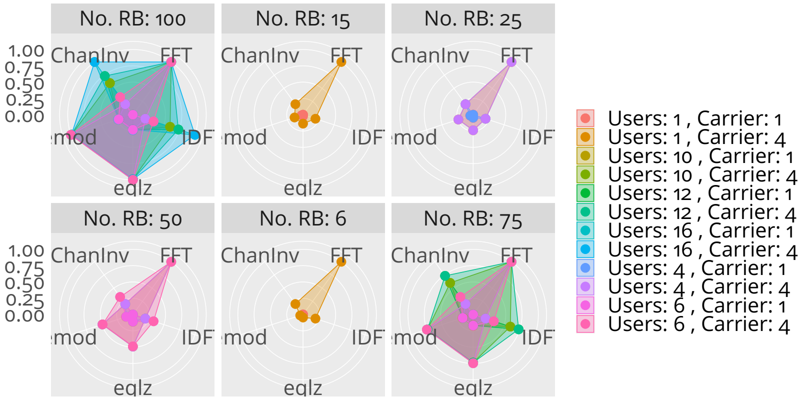


Figure 6.8: Different parameter combinations and their effects on the requirements on computation in LTE. "No. RB" denotes the number of resource blocks.

band processing and how the required computations depend on them. In the figure, the number of antennas and layers are both fixed at 1. These also affect the required numbers of FFT, IFFT, equalization, demodulation and channel inversion nodes. The length of the values in the radar plot shows the relative amount of kernels required for each type of operation.

Even for fixed parameters, designing modems for 5G is already an extremely complex endeavor on its own. In order to make a system adaptable, developers need to ensure that the behavior remains correct when adapting. A model that ensures deterministic execution for easy debugging and time semantics for real-time reasoning is thus sought for this type of applications. For this reason, we propose to use the Reactors model [Wit+20]. We adapted the WiBench benchmark [Zhe+13] (cf. Figure 6.10), dealing with PHY in LTE, using Lingua Franca [Loh+20a]. Lingua Franca is an implementation of the Reactors model.

Adaptability in 5G and beyond

We used LTE traces to extrapolate information about the dynamicity of the demands for 5G and beyond. These traffic traces, collected over a 5 hour period spread over 15 days feature real data with over 1.2 million Radio Network Temporary Identifiers (RNTIs) from 24 different base stations [BCM20]. They were generously shared by Arka Maity, Nishant Budhev and Tulika Mitra.

Figure 6.9 shows statistical data points extracted from the traces. In the figure, every point represents the workload of a base station at a particular set of subframes. We consider the relationship between the number

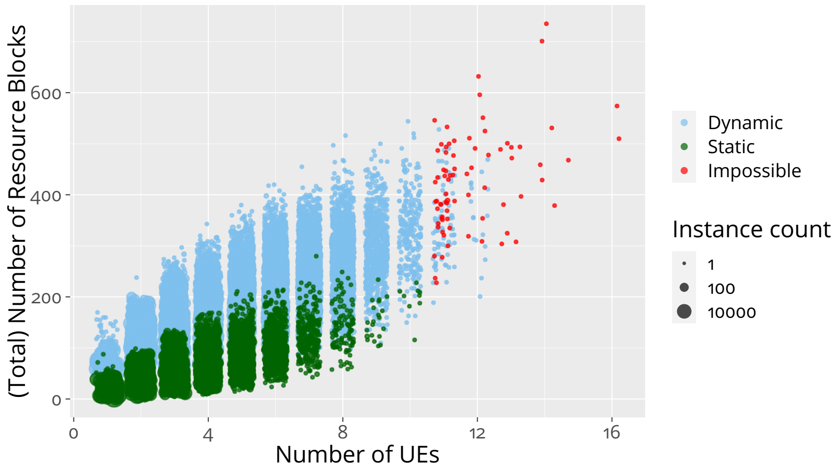


Figure 6.9: Possible configurations in a resource-constrained LTE environment. The number of UEs are depicted with a meaningless random jitter for visibility. Adapted from Figure 2 in [Wit+20].

of User Equipment (UE) units and the total number of resource blocks required at those subframes. The size of each point represents how many subframes in the trace had those precise requirements in terms of UEs and total resource blocks.

In a traditional setup, with a static implementation of a PHY, we would make worst-case assumptions. This includes the number of PEs and resource blocks per UE that we can support, as dictated by resource constraints. Let's assume that, given our limited resources, our PHY implementation supports at most 10 UEs and 47 resource blocks per UE. These numbers are admittedly low, even for LTE. However, we choose this low threshold deliberately. This way we can use these LTE traces to extrapolate the possible dynamicity of behaviors in 5G and beyond. Out of the 3017424 considered subframes in the traces, slightly over half (1689447) could be processed with these static resource constraints. These design points we classify as "Static" and depict as green points in Figure 6.9. Consider a dynamic system that can adapt to the current workload, e.g. by supporting less UEs but more resource blocks per UE, or vice-versa. We can estimate the resources required for different configurations by generating task graphs for each and comparing the computational resources required. Some design points in Figure 6.9 can be supported by a dynamic system using the same resources as in the static version. We classify them as "Dynamic". Slightly less than half of the subframes in the trace (1327900) fall under this category. Finally, the remaining 77 points would need more resources to be supported, even with a dynamic system. We classify them as "Impossible" (red points). Overall, this means that over 99.997% of the subframes observed could be implemented by a dynamic system, where a static one using roughly the same resources covers less than 56%.

This statistical analysis of LTE traces shows very clearly how much a baseband system could benefit from dynamically adapting to the workload. We can cover significantly more design points with constrained resources. Equivalently, we can use less resources to cover all the observed cases. However, such dynamic systems come with several drawbacks. Programming base station decoders is already a complex endeavor for static systems, much more so with dynamic ones. With real-time require-

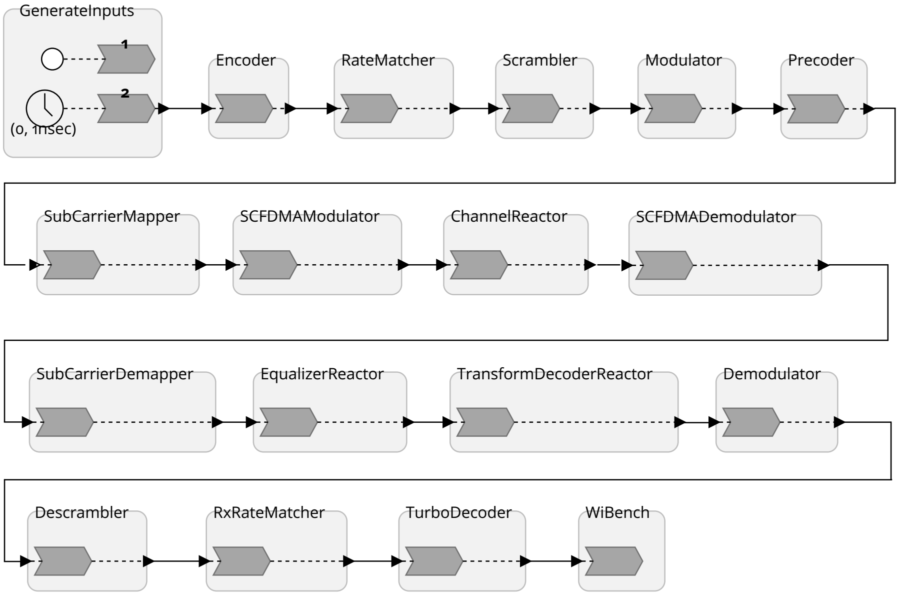


Figure 6.10: The Reactor network of the modified WiBench benchmark in Lingua Franca.

ments, we must ensure that the changing system not only respects the deterministic semantics of the decoder, but also the timing requirements. This is why we propose to use a formal model of computation to describe 5G (and beyond). Using the model of Reactors we can make the execution deterministic and timed. It also can help define well-behaved dynamic behavior through the use of mutations in future work.

Modeling 5G with Reactors

In ongoing (unpublished) work with Robert Wittig and Christian Menard, we adapted the WiBench benchmark [Zhe+13] to work with Lingua Franca, an implementation of Reactors. Figure 6.10 depicts the Reactor network implementing this benchmark. Since WiBench is single threaded, we only compared to a single threaded version in Reactors. In particular, we did not leverage data level parallelism throughout the layer, nor the pipeline parallelism that we get from the network's topology for free. This is a worst-case assumption we made to analyze the overhead. By using the Reactor model, the benchmark is deterministic, even if it was to run using this parallelism [Loh+20c]. More importantly though, we can use the model's time semantics to define the constraints that ensure each subframe is processed on time. Our implementation is thus still static (cf. Figure 6.10), since we have not yet specified well-defined mutations. This implementation presents a great opportunity for future work to research and develop safe mutations for 5G.

Our implementation of the Reactor-based WiBench had an overhead of 15% (median over 100 executions), compared to the baseline implementation of WiBench. There is certainly potential to improve this, e.g. as the scheduler of the C++ implementation on Lingua Franca, used for this implementation, was not optimized at all. Nevertheless, this is a purely software-based implementation, so it serves only as a very rough estimation of the overhead; it is best suited to study the model's suitability and develop Reactor mutations for adaptability. An efficient implementation in practice could work with reconfigurable hardware, e.g. implementing a Precision Timed (PRET) [EL07] machine, which is well-suited to Reactors' semantics.

In general, these preliminary results open up many avenues for research in adaptability in 5G. We can the Reactors model, at the semantic level to support the necessary adaptability in 5G. Similarly, we can design reconfigurable hardware that implements it.

Other applications: Automotive

The reactors model has many desirable properties for designing reliable CPSs, which can be applied in a multitude of domains. An important example is the automotive domain, where the high-performance requirements of autonomous driving and modern entertainment are coupled with the timed CPS including the car and its surroundings. To keep the scope of this thesis limited, we omit a thorough discussion of an application of Reactors in the automotive domain. In [Men+20], we showed how we can use the Reactors model to achieve determinism in the AUTOSAR Adaptive Platform (AP), a modern automotive standard.

7

PROGRAMMING LANGUAGES

In this thesis we have discussed multiple Models of Computation (MoCs), reasoning about their semantics and how to best deploy them on a particular hardware architecture. A natural question arising from this is, “how do we program in these MoCs?”.

In Chapter 2, Section 2.1 we saw the C for Process Networks (CPN) language. It is a DSL designed to describe data flow programs with the KMQ blocking-read semantics, with special annotations for SDF actors. Other MoC-based languages exist, like the CAL actor language [EJ03], or Lingua Franca [Loh+20b]. These languages allow “freedom from choice” [Lee19], by enforcing a model that limits the ways in which to make mistakes, ideally without compromising the expressiveness of what can be designed with the model.

A common trade-off when designing programming languages is also the question of expressiveness versus performance. High-level expressive abstractions are often at odds with low-level performance optimizations. However, well-designed abstractions can use semantics-preserving compiler transformations to still derive an efficient execution. The whole principle of software synthesis can be seen as an instance of this.

This chapter discusses programing languages for defining and enforcing the semantics of a MoC. After a short review of existing languages, it focuses on the Ohua [Ert19] language, which defines dataflow implicitly. It also discusses how we can leverage the language and its semantics to define semantics-preservings transformations at a language level. We show this for a use-case optimizing I/O on microservice-oriented architectures.

7.1 Freedom from Choice

This section reviews some programming languages and how they provide “freedom from choice” in the sense of A. Sangiovani-Vincentelli [Lee19]. There is a distinct sense in which this is the central question of programming languages in general. By removing memory management through having no pointer arithmetic and garbage collection, Java frees its users from multiple families of errors that are possible in C. Rust’s ownership types take a different approach, also removing complete families of memory-management based errors, without introducing large performance overheads or unpredictable behavior from the garbage collector. Elm [Cza12], on the other hand, exposes a functional paradigm with a strong type-system for GUI development of web applications, which eliminates virtually all run-time errors.

These kinds of “freedom from choice” are beyond the scope of this thesis, which focuses on MoCs like those described in Chapter 6. In more constraining MoCs, like the ones discussed here, the temptation to break away from the semantics might be higher. An interesting observation and discussion of this phenomenon can be found in [TDJ13]. Not only does it show that developers commonly break away from the semantics if these are not enforced, but also gives multiple explanations why. This is why we believe MoCs should not be exposed as a library, but rather as implicit in

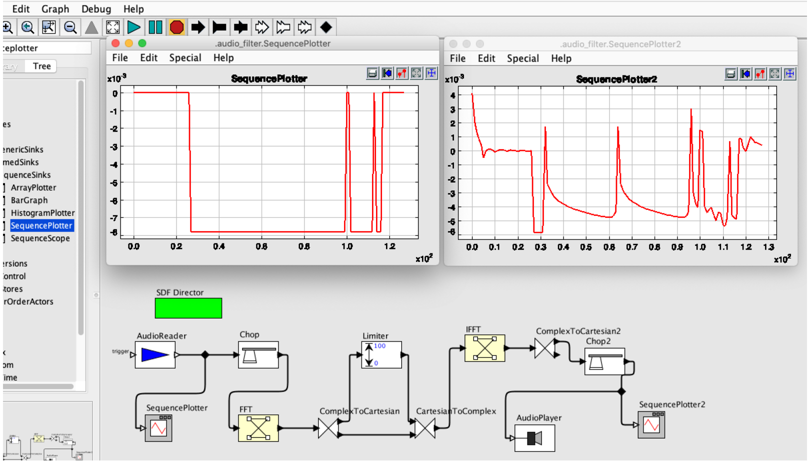


Figure 7.1: An audio filter in SDF semantics in Ptolemy II

the semantics of a (full) language. Here we will briefly survey languages focused on MoC-based paradigms.

7.1.1 *Dataflow, Actors and Discrete Events*

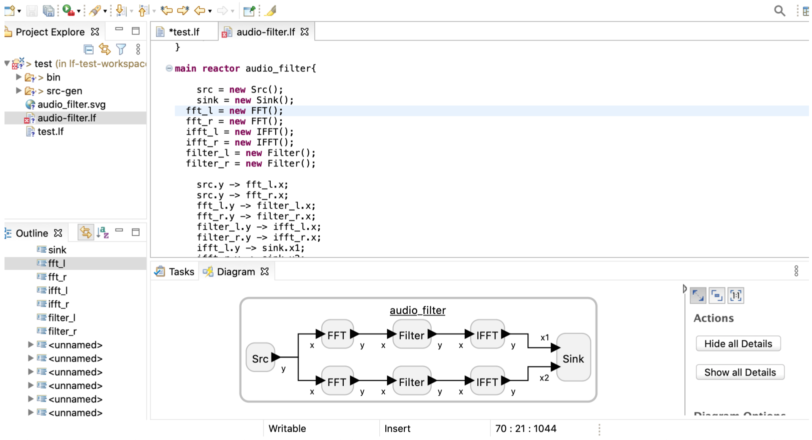
Most well-known languages that follow MoCs like the ones described here are based on the actor model. Compared to more sophisticated models, the actor model has mostly intuitive semantics. The Erlang language is a successful example of a language whose semantics are in principle an implementation of the (Hewitt-Agha) actor model. Another example of an actor based language is Rebeca [Sir04], which is primarily a design and specification language used in model checking. Another prominent example is the CAL actor language [EJ03]. It has been used to define the Reconfigurable Video Coding (RVC) standard [Bha+11]. The RVC-CAL compiler[1] is an Eclipse-based compiler for the CAL actor language, which is used to compile the RVC reference implementations.

Ptolemy II [Pto14] is another prominent MoC-centered programming environment. In contrast to most of the other frameworks, Ptolemy II supports a plethora of MoC, including most of the models discussed in this thesis. Also in contrast to most other frameworks, MoCs are a central component of Ptolemy II, which makes them explicit using *directors*. The framework uses the Java programming language to allow the definition of arbitrary actors, but it also comes with a large library of pre-defined actors. Figure 7.1 shows an implementation of an audio filter, which while not identical, is semantically similar to our running example described in Chapter 2. This implementation uses only pre-defined actors in Ptolemy II.

In the case of discrete-event models, there are many well-established languages which implement them. The hardware description languages VHDL and Verilog work with discrete-event semantics, since hardware arguably does. The SystemC language, well known for discrete event simulations (of hardware), also has discrete event semantics [Mue+01]. Similarly does the related SpecC language [Gaj+12]. Finally, more on the software side are the Synchronous languages, like LUSTRE [PHP87] or ESTEREL [BD91], which are (complete) programming languages with discrete-event semantics.

---

[1] `https://sourceforge.net/projects/orcc/`



```
    }

main reactor audio_filter{

    src = new Src();
    sink = new Sink();
  fft_l = new FFT();
  fft_r = new FFT();
  ifft_l = new IFFT();
  ifft_r = new IFFT();
  filter_l = new Filter();
  filter_r = new Filter();

    src.y -> fft_l.x;
    src.y -> fft_r.x;
    fft_l.y -> filter_l.x;
    fft_r.y -> filter_r.x;
    filter_l.y -> ifft_l.x;
    filter_r.y -> ifft_r.x;
    ifft_l.y -> sink.x1;
```

Figure 7.2: The audio filter example in Lingua Franca

The Lingua Franca language [Loh+20b][2] is also in the discrete events domain. Lingua Franca is a complex framework that implements the Reactors model, described in Section 6.3. This novel language is a self-described polyglot *coordination language*, which means that it is not used to define the computation, but rather, to compose reactors written in a different language. Nothing prevents programmers from "cheating" in the code of the reactors and going around the semantics, yet it does enforce them more strongly than a library. Lingua Franca is a (complete) DSL that generates Reactors-based applications in a source-to-source compilation process. Figure 7.2 shows the Eclipse-based programming environment of Lingua Franca, with an implementation of the audio filter benchmark. We will not discuss the framework more in detail, as the design and implementation of Lingua Franca is not part of the contribution of this thesis.

In Chapter 2 we discussed the CPN language, as well as the MAPS framework which is used to lower CPN to different implementations in heterogeneous systems. In contrast[3], e.g. the YAPI programming interface [Koc+00] defines process networks only as a runtime library and can be evaded just like Scala developers do with actor frameworks [TDJ13]. The Sesame framework uses a DSL called YML, and also supports YAPI-based programs, e.g. derived from Compaan [SD03] for DSE[PEP06]. Most of the software synthesis flows discussed in Section 2.6 use the languages described here, e.g. TURNUS which uses CAL or SystemCoDesigner which is based on SystemC. Other systems like DAARM or `mocasin` do not use actual source code for the applications but rather application models for DSE.

Most of the flows discussed above and in Section 2.6 are academic and deal with more sophisticated MoCs. Many ideas that seem good in academia do not hold up in a practical development environment, where the learning curve of the models and time-to-market considerations change the field. It is therefore not surprising that the more sophisticated models have seen less adoption. Nevertheless, some MoC-based commercial systems do exist and have seen successful adoption. For example, the LabVIEW Communications System Design Suite restricts the LabVIEW language to a dataflow MoC. Similarly, Matlab Simulink has a

---

[2] https://github.com/icyphy/lingua-franca

[3] As an extension of C, the CPN language also does not fundamentally prevent programmers from breaking the semantics.

```
(defn -main
"Audio filter example"
[args]
(ohua
  (smap
    (algo [s]
          (let [[x y] (split s)]
            (let [[xout yout]
                  [(ifft (filter (fft x)))
                   (ifft (filter (fft y)))]]
              (sink xout yout))))
    (src))))
```

Listing 5: The Audio Filter Example written in Ohua

dataflow-like semantic with time triggering, and is thus closer to discrete event models. On the discrete events side, as mentioned above, hardware description languages like Verilog or VHDL have discrete-event semantics. These languages are very widespread and are used commercially.

7.1.2 *Implicit Dataflow*

The languages surveyed so far are explicit about their abstractions: Actors, Reactors or Processes are declared explicitly. Similarly, channels describing the data flow are made explicit either through channel declarations or through the connection of explicit ports. A programmer writing in e.g. CPN or Lingua Franca has to have a model of the network describing the application in their head (or in their IDE). Implicit abstractions, on the other hand, work by generating implicit models from linguistic constructs that don't exhibit their structure directly.

Implicit abstractions, as we just defined them, are ubiquitous in programming languages. Objects in object-oriented programming (OOP), for example, are an implicit abstraction for data encapsulation that is fundamentally similar to actors. A thorough classification of these implicit models is outside the scope of this thesis. Instead, we will look closely at the Ohua programming paradigm [Ert19], which derives a dataflow execution from functional semantics.

The Ohua programming paradigm, by S. Ertel, and others is an implicit model of concurrency. It can be used to express concurrency at a language level, without explicit constructions, like threads and locks. This comes from lowering an Ohua program into a dataflow-based execution. This model is not part of the original contribution of this thesis. We will introduce it here as background material.

Ohua itself is a general paradigm that works on multiple languages, and the framework has evolved over the years of its development. The version of Ohua we will discuss here is based on Clojure and Java, but the Ohua compiler and its principles work with many languages. Rutimes also exist for Rust, Javascript or Go, at different levels of maturity. Ohua is best understood by diving directly into examples.

Consider the code in Listing 5. The code in the example is written in a DSL embedded in Clojure, a dialect of Lisp. It implements the same example from Chapter 2 (cf. Listing 2 or Figure 2.1), a two-channel audio filter. Internally, the compiler transforms this code into a dataflow graph (similar to that depicted in Figure 2.1) for execution. A special function, `ohua`, annotates the AST it receives as argument to be executed as implicit dataflow. The `smap` function is a special variant of `map` that considers state in the functions. We will discuss the semantics of `smap` in Section 7.2. Finally, the `algo` definition in Ohua is akin to the anonymous function definition `fn` in Clojure. It defines Ohua "algorithms", which are transformed to dataflow actors. As a MoC, this can be embedded in the Dennis dataflow models discussed in Chapter 6.

The example in Listing 5 can be transformed into a dataflow graph for execution.The main advantage of this transformation is that a dataflow graph exposes concurrency, which can be exploited e.g. in a parallel execution or for optimizing I/O (cf. Section 7.3). This duality between code and dataflow graphs is a core concept behind Ohua. The other central pilar of the Ohua design concept are stateful functions, an abstraction that encapsulates functions with state and side-effects in the context of their dataflow execution.

7.1.3 *Stateful Functions*

The functional programming community has made the distinction between *pure* and *impure* functions widespread. A pure function is a function in the mathematical sense of the word: it receives a certain input and, deterministically, produces an output. This could be as simple as negating a boolean value, or as complicated as inference with a gargantuan deep neural network. The main point is that the entirety of the usage of a function is that it returns a value in a deterministic fashion from its inputs.

In most imperative languages, like C or Java, functions usually also have side-effects. Writing the output to the terminal, storing data in a global data structure or even reading data from a sensor in a CPS, these are all examples of side effects. A language that only allows pure functions is basically useless, since even printing the result of a computation is impure.

Stateful functions are a special abstraction, where the concept of pure functions is extended to consider the state of the computation. While this excludes aspects like the time of the computation and side-effects like actuation, it is general enough to cover large classes of functions used in most software. A stateful function is a function $f : a \to b$ and an abstract state S, where the execution of the function can be seen as dependent on the state, which it also modifies. In other words, we consider f as a function:

$$f : a \times S \to b \times S \tag{7.1}$$

Pure functions can be seen as a special case of stateful functions, with a trivial state $S = \{*\}$. Listing 6 shows an example of a stateful function, written in Java, which is identified as such by the `@defsfn` annotation [EAC18]. It models a parser, which writes the parsed symbols to a symbol table. The table, a private object of the class, is the state of this stateful function. It is implicitly managed as the state by the Ohua runtime.

7.2 Stateful Parallelism

Software synthesis for multicores is effective because we use MoCs that expose concurrency. The most natural way of leveraging the exposed con-

```
public class ParseVariable {
  // state
  private SymbolTable table;

  @defsfn
  public SymbolObject parse(ExpressionObject expr) {
    var symbols = expr.parse();
    this.table.write(symbols);
    return this.table.getLastSymbol();
  }
}
```

Listing 6: An example of a stateful function.

currency is parallelism. When writing code using explicit models in the dataflow family, like KPN or SDF, the concurrency also becomes explicit. However, in an implicit language like Ohua, we need to be careful to consider stateful computation when extracting concurrency.

Ohua uses a special operator, `smap`, to derive concurrency from stateful computation. The principle behind `smap` is that it extends the higher-order `map` function to consider the state of the function it maps. This adds a dependency between multiple executions of the same (stateful) function. For a single function mapped over a collection, this principle exposes no concurrency. There is none, in general. However, when we compose multiple functions in a `map`, we get a different picture.

Consider three (stateful) functions, $f : a \to b$ and $g : b \to c$, $h : c \to d$ with respective states S_f, S_g and S_h, which we thus model as functions:

$$sf : a \times S_f \to b \times S_f, sg : b \times S_g \to c \times S_g, sh : c \times S_h \to d \times S_h$$

We use the prefix `s` to distinguish the stateful version of the function (with the state dependencies explicit). Then, we get the dependency graph for the execution of the (Haskell) expression `map (f.g.h) inputs` as depicted in Figure 7.3

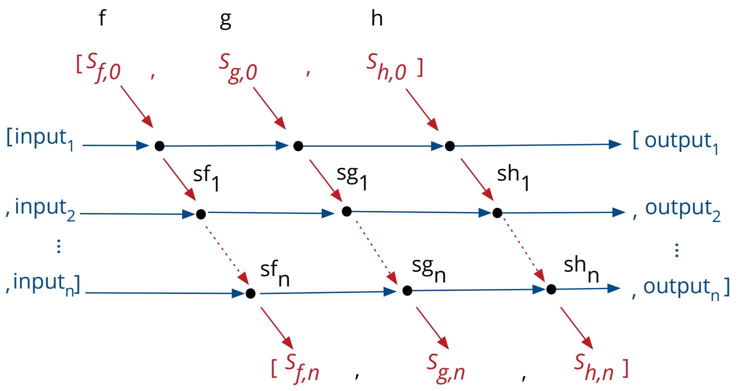


Figure 7.3: Dependencies of (`map (f . g . h) inputs`). Adapted from Figure 5 in [Ert+19b]

The pattern we see in Figure 7.3 is very similar to the higher-order function `scan` on the state. Intuitively, if we consider the function `sf'` that only returns the state of `sf`, then the state $S_{f,i} := (S_f)_i$ corresponds to the

i-th value of the expression `scanl sf'` $S_{f,0}$ `input`. Threading the state around `f` explicitly can be achieved with the functional pattern known as a monad, in a fashion similar to the state monad in Haskell. Two different concrete implementations of this principle in Haskell are discussed in [Ert+19b]. These implementations are beyond the scope of this thesis. The result of this monadic composition of state threads, however, is that we can write virtually the same expression as above in a monadic composition:

```
smap (f >=> g >=> h) inputs
```

The >=> operator is the monadic equivalent of function composition, with the . (dot) operator. This yields the dependencies explicitly. Similarly, these state threads and their composition can be formalized using category theory [Ert+19a]. This formalization is rather technical. We will only sketch it here.

Let $\mathcal{C}$ be a Cartesian closed category, which is a technical condition that in a model-theoretic interpretation of categories corresponds to the typed λ calculus [Hue85]. We can think of $\mathcal{C}$ as the values and functions of the language, like the category of Haskell types Hask[4]. A Cartesian closed category has a terminal object $\perp \in \mathrm{Obj}(\mathcal{C})$, and any two objects $B, C \in \mathrm{Obj}(\mathcal{C})$ have a product $B \times C$ and an exponential B^Y. These constructions are defined via universal properties in commutative diagrams and are rather technical. We will omit the precise definitions here for space reasons. It suffices to say they correspond with the known constructions, e.g. the product is the Cartesian product in the category Set of sets.

The main idea of formalizing and dealing with state threads is to index them. We do this through a (countable) index set $N \subseteq \mathbb{N}$, which for practical purposes we can also think of as being finite. We "split" the state into local states which correspond to the indices in N. Formally, let $S_i \in \mathrm{Obj}(\mathcal{C}), i \in N$ be pairwise distinct (i.e. $i \neq j \Rightarrow S_i \neq S_j$). We define for $I \subseteq N$ the *state object* $S_I = \times_{i \in I} S_i$ as the product of the S_i for all $i \in I$. If $I = N$, we call the state object S_N the *global state*. The individual states $S_i := S_{\{i\}}$ for $i \in N$ we call fundamental states. We thus formally define a *state thread* as a morphism:

$$f : (a \times S_I) \to (b \times S_I), \text{ for an } I \subseteq N$$

This definition formalizes the intuition behind Equation 7.1. It is justified by Lemma 7.2.1.

Lemma 7.2.1 (Lemma 1.3 of [Ert+19a])**.** The following define the objects and morphisms of a subcategory $\mathcal{S}$ of $\mathcal{C}$,

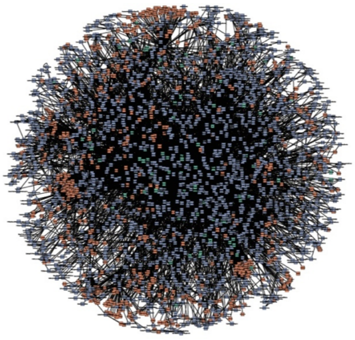
$$\mathrm{Obj}(\mathcal{S}) = \{a \times s_I \mid a \in \mathrm{Obj}(\mathcal{C}), I \subseteq N\}\ , \tag{7.2}$$

$$\mathrm{Morph}(\mathcal{S}) = \{f : (a \times s_I) \to (b \times s_I) \mid f \in \mathrm{Morph}(\mathcal{C}), I \subseteq N\}\ . \tag{7.3}$$

Proof. Since $\mathcal{C}$ is a category, and as such the composition of morphisms behaves as required, it suffices to show that morphisms respect the structure of the subcategory. It is clear that $\mathrm{id}_{a \times s_I} \in \mathrm{Morph}(\mathcal{S})$ for every $a \in \mathrm{Obj}(\mathcal{C}), I \subseteq N$, since $\mathrm{id}_a \in \mathrm{Morph}(\mathcal{C})$. Let $f, g \in \mathrm{Morph}(\mathcal{S})$ with such that $g \circ f$ is defined in $\mathcal{C}$. Then it has to hold that there are a, b and $c \in \mathrm{Obj}(C)$ as well as $I \subseteq N$, such that $f : (a \times S_I) \to (b \times S_I)$ and $g : (b \times S_I) \to (c \times S_I)$, since $g \circ f$ is defined in $\mathcal{C}$. But then $g \circ f : (a \times S_I) \to (c \times S_I)$ is in $\mathrm{Morph}(\mathcal{S})$ by definition of $\mathcal{S}$. □

---

[4] Note that this might not, strictly speaking, be a category. See `http://math.andrej.com/2016/08/06/hask-is-not-a-category/`



[Source: I Love APIs 2015 by Chris Munns, licensed under CC BY 4.0, available at: `http://bit.ly/2zboHTK`]

Figure 7.4: Microservices at Amazon.

Intuitively, we think of a state thread $f : (a \times S_I) \to (b \times S_I)$ as operating only on the state object S_I, which is a part of the global state S_N. As can be seen from the proof of Lemma 7.2.1, state threads compose when they have the same state objects S_I. The goal of the formalism is to be able to extend this composition to arbitrary $I, J \subseteq N$. In particular, this gives us a framework to reason about the dependencies of state threads. Indeed, if $I \cap J = \varnothing$, then the state threads $f : (a \times S_I) \to (b \times S_I)$ and $g : (c \times S_J) \to (d \times S_J)$ are concurrent.

The definition of a general composition of state threads is, again, somewhat technical. We will only sketch it here: Given a state thread $f : (a \times S_I) \to (b \times S_I)$, the idea is to define an extension $f* : (a \times S_N) \to (b \times S_N)$ which acts as an identity on $a \times S_{N \setminus I}$. By using a technical construction, this extension can be made such that the information of the I is kept, while allowing a well-defined composition of state threads. In this way, the resulting category allows us to elevate arbitrary computations in $\mathcal{C}$ to state threads in a fashion that allows us to reason about their composition by looking at the state objects S_I, as described above.

The formalism outlined here defines the semantics of `smap`, allowing us to express implicit concurrency even when dealing with stateful functions. With the Haskell implementation from [Ert+19b], this allows us to extract implicit parallelism. These `smap` semantics of Haskell are the same as the Clojure-based `smap` in Ohua. The implicit concurrency extracted with Ohua, however, enables more than a parallel execution. We can also use it to optimize I/O.

7.3 Concise code and Efficient I/O

Dataflow models expose concurrency. While concurrency enables parallelism, it can also be leveraged in other ways. This section discusses Ÿauhau [Ert+18], a framework for optimizing I/O in microservice-based systems. This Ohua-based framework uses the concurrency exposed by the MoC to do so.

The infrastructure of large internet companies today relies to a great extent on microservice-based architectures [DeC+07; Mar+14]. Figure 7.4 shows the microservice infrastructure of Amazon circa 2015. It shows microservices as nodes, and how they depend on each other as edges. The figure serves to illustrate the complexity of microservice-based architectures.

In microservices, I/O plays a crucial role in the performance. A microservice will commonly send multiple requests to a different microservice, as part of its operation. Each request comes with a significant overhead from establishing the connection and sending the data. If the requests do not depend on each other, however, they can instead be sent as a single, batched request for mitigating the overhead.

Batching requests is not a novel idea, it is well-established as a technique to optimize I/O. The trade-off comes from the code required to write batched requests. A developer writing code for such a microservice-based architecture needs to take both the functionality and the I/O optimizations into account. This makes the code harder to write, read and maintain, as the optimizations clutter the readability of the functionality. This is unacceptable in a context where development time is a highly valuable resource, be it for human-resource costs or because it is important to have a working solution as quick as possible.

The situation described is the case for Facebook's spam-fighting services [Mar+14]. When fighting spam, a novel filter must not only be effective and perform efficiently, it also should be implemented as fast as possible without compromising the functionality. An ideal spam-fighting system thus allows the developers to focus on the functionality, and optimizes the implementation without cluttering the code. This is precisely what the Haxl system attempts, using the Haskell abstraction of *applicative functors*. Consider the Haskell code snippet (taken from [Mar+14]) in Listing 7:

```haskell
let numCommonFriends =
     length (intersect (friendsOf x) (friendsOf y))
in
if numCommonFriends < 2 && daysRegistered x < 30
    then...
    else...
```

Listing 7: An example of a Spam-fighting request to be optimized by Haxl (from [Mar+14]).

The listing shows a code example where, for spam fighting, the number of common Facebook friends of two users are calculated. This is done by calling the function `friendsOf` for both `x` and `y`. Code like Listing 7 is easy to read, but unoptimized, it will send to requests to the microservice that handles the `friendsOf` function. The solution of Haxl is to use applicative functors to compose I/O calls, such as `friendsOf`, and automatically batch independent requests this way. Listing 8 shows how this is achieved in Haxl using applicative-do notation.

Under the hood, the applicative functor definition for `friendsOf` in Haxl gathers the arguments `x` and `y` and makes a single, batched request. This optimizes the execution with a minimal obfuscation of the code: A devel-

```haskell
do a <- friendsOf x
   b <- friendsOf y
   return (length (intersect a b))
```

Listing 8: The request from Listing 7 batched using applicative-do (from [Mar+14]).

$$\begin{array}{l} \texttt{let}\ \texttt{numCommonFriends} = \\ \quad \texttt{ff}_{\text{length}}(\texttt{ff}_{\text{intersect}}(\\ \qquad \texttt{ff}_{\text{friendsOf}}(\texttt{io}(x)),\ \texttt{ff}_{\text{friendsOf}}(\texttt{io}(y)) \\ \quad))\ \texttt{in}\ \ldots \end{array}$$

Listing 9: The request from Listing 7 in Ÿauhau.

oper has to switch from a pure functional style to an applicative style, but can otherwise focus on the semantics of the program.

Our central observation is that this I/O optimizations all come “for free” from the exposed concurrency in a dataflow execution. If we define a dataflow operator that gathers the inputs and issues a single batched request, we get the composition from the semantics of dataflow. This is the main idea behind Ÿauhau.

Ÿauhau is based on the iteration of Ohua embedded in Clojure. It is a language with an explicit annotation for functions which perform I/O. Leveraging these annotations, a semantics-preservig transformation can minimize the number of independent I/O-performing function calls. We map the Clojure-based Ohua to an internal expression IR, as is shown in Figure 7.5.

$$\begin{array}{rcl@{\qquad}rcl} \texttt{x} & \mapsto & x & \texttt{(let [x t] t)} & \mapsto & \texttt{let}\ x = t\ \texttt{in}\ t \\ \texttt{(io x)} & \mapsto & \texttt{io}(x) & \texttt{(fun [x] t)} \equiv \texttt{\#(t)} & \mapsto & \lambda x.t \\ \texttt{(t t)} & \mapsto & t\ t & \texttt{(f x1 ... xn)} & \mapsto & \texttt{ff}_f(x_1 \ldots x_n) \\ & & & \texttt{(if t t t)} & \mapsto & \texttt{if}(t\ t\ t) \end{array}$$

Figure 7.5: Mapping the terms of the Clojure-based language to an expression IR. Adapted from Figure 9 in [Ert+18].

The expression IR defined in Figure 7.5 is based on λ calculus with a `let` construction for lexical scoping and explicit conditionals (`if`). The central innovations of the language are the two particular terms, `ff` for foreign functions, which is any (possibly stateful) function. The other term is `io`, which is an explicit annotation that a function does I/O. The premise is to optimize the number of annotated I/O calls leveraging the concurrency from the dataflow semantics derived from this language. The example in Listing 9 lists the same request as Listing 7 in Ÿauhau.

The I/O optimizations in Ÿauhau come from batching requests. Instead of calling $\texttt{ff}_{\text{friendsOf}}(\texttt{io}(\texttt{x}))$ and $\texttt{ff}_{\text{friendsOf}}(\texttt{io}(\texttt{y}))$ as two separate I/O requests, we can call them as a single batched request, since they are independent. In Ÿauhau we do so by introducing a *batched* I/O statement, `bio`. The `bio` statement takes a list of arguments and does a batched call with this list. In the example, the two statements become a single $\texttt{ff}_{\text{friendsOf}}(\texttt{bio([x,y])})$.

Through semantics-preserving transformations using `let` floating we can get independent I/O calls batched this way and transform them to dataflow. The explicit concurrency in the dataflow transformation allows to execute batched I/O calls when they arrive. The details of the transformation [Ert+18] are beyond the scope of this thesis. We discuss how we

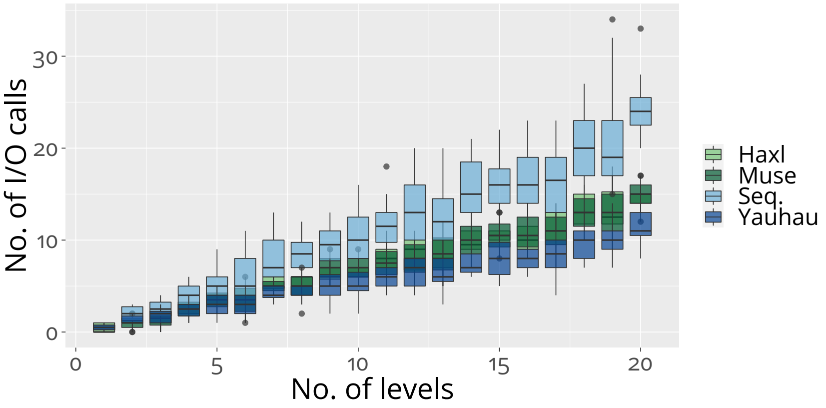


Figure 7.6: Batching I/O with Ÿauhau compared to Haxl and Muse. Adapted from Figure 11 of [Ert+18].

evaluate the effectiveness of the transformation using the Level Graphs methodology described in Section 3.3.

Figure 7.6 shows a comparison of our framework, Ÿauhau, with the two other main alternatives in the public domain, Facebook's Haxl [Mar+14] and Muse [Kac15]. We cannot compare with Twitter's Scala-based variant, Stitch[Don14], which is neither described in a peer-reviewed publication nor open-sourced. We do not have a real test workload, like Facebook does, to reproduce the evaluation of [Mar+14]. For this reason, we used synthetic graphs with a similar structure to compare the approaches. We produced random Level Graphs with code annotations as described in Section 3.3. We varied the number of levels between 1 and 20 in the generated graph. An increasing number of levels results in increasingly complex applications and interactions. For this baseline comparison we disabled sub-functions by letting the probability of labeling a node as a sub-function be 0. For each number of levels, we generated 20 graphs and produced code for each of the frameworks with the structure of each of these twenty graphs. This way, the same workload is evaluated for each framework, as equivalent versions are produced from a single graph. The number of I/O accesses in each individual graph is random nevertheless, which is why we used 20 graphs for each number of levels considered, to obtain a statistical assessment of the batching effect of each framework.

The results of Figure 7.6 show that both Haxl and Muse consistently batch I/O calls, improving over a sequential execution in a statistically significant fashion. The ratio of I/O calls compared to a sequential execution improves with increasing code complexity, as measured by the number of levels. Compared to Haxl and Muse, which produce equivalent results, Ÿauhau batches significantly more I/O calls. The reason for Ÿauhau beating the other two frameworks is that it can batch across multiple levels, due to the underlying dataflow representation. Both Haxl and Muse work in *rounds*, which depend on the code structure. Thus, to fully optimize I/O in these frameworks, developers need to re-organize their code to maximize batch requests in each round. This partially defeats the purpose of the DSL, namely to free developers from performance concerns and let them focus solely on the functionality.

Figure 7.7 shows a second experiment, where concurrency in I/O is enabled in the dataflow graph. Since Haxl and Muse are equivalent in terms of their batching, we just compare to Haxl in this experiment. We com-

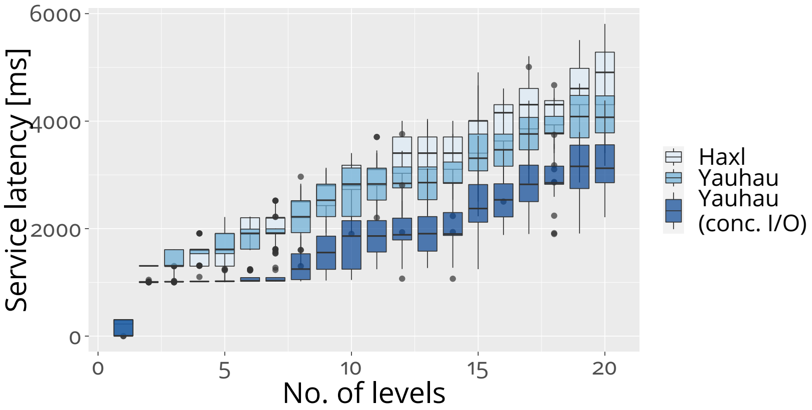


Figure 7.7: Concurrent I/O with Ÿauhau compared to Haxl and Muse. Adapted from Figure 12 of [Ert+18].

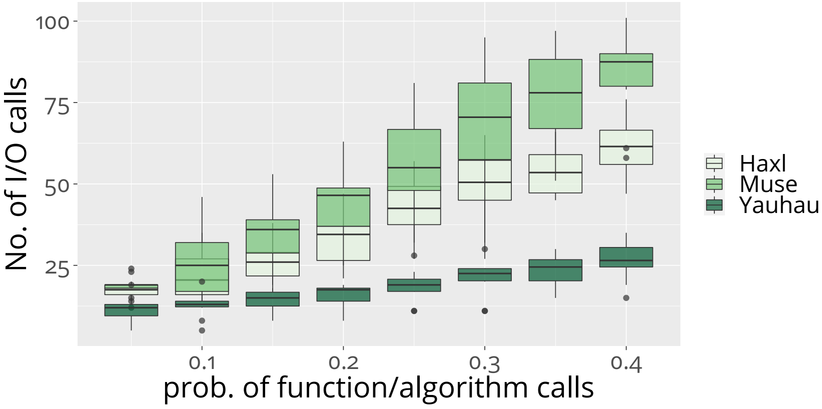


Figure 7.8: Concurrent I/O in modular programs with Ÿauhau. Adapted from Figure 13 of [Ert+18].

pare Ÿauhau to a variant which blocks computation when performing I/O and a concurrent dataflow variant. Again we use level graphs to produce benchmarks tailored to our problem, this time we add a fixed (randomly-chosen) execution time for the computation and I/O nodes in the graph, to simulate the computation time and I/O latency. We see how the concurrency of the dataflow MoC improves the latency of the service significantly beyond the additional benefits already provided by the round-free batching.

Finally, an important advantage of the dataflow MoC we have not evaluated is how it also allows us to transcend the function boundaries. To test this we can again leverage the full control we get from our Level Graph benchmarking method. We fix the number of levels, controlling the code complexity, and consequently the average number of I/O requests. Instead of varying the number of levels, we allow sub-functions and vary the probability of generating such a sub-function. The higher the probability, the more nested functions appear in the code. In other words, we can make code with similar levels of complexity be more or less modular. This allows us to test directly how modularity affects batching.

Figure 7.8 shows the results of the modularity experiment. For this case, Haxl and Muse differ, since our Haxl back-end uses applicative-do desugaring, which allows the compiler to optimize more than in Muse. Most notably however, both Haxl and Muse significantly increase the amount of I/O requests when the program becomes more nested, while in Ÿauhau this is not the case. This is because of the underlying dataflow MoC, which flattens the dependency graph, effectively inlining the sub-function calls and allowing the framework to batch across function boundaries. In particular, this means again that the premise of making developers not worry about performance in the DSL is partially broken by Haxl and Muse, and this is fixed by Ÿauhau.

We have seen how MoC-based design can be useful in a different context, besides the CPS applications we focused on in the previous chapters. In particular, the Ohua approach can be used to make the model more implicit in the computation, which is crucial for developer adoption in many contexts. Finally, we have seen in this section as well some concrete advantages of random benchmarks with a clearly-defined structure, in the concrete example of the Level Graphs from Section 3.3. They allowed us to tailor experiments to isolate specific features of our dataflow MoC-based approach and compare them to the other state-of-the-art approaches, which would not have been possible otherwise.

8

RELATED WORK

The nature of this thesis is not focused enough for us to discuss related work from a general point of view. Perhaps the closest in spirit to the work presented in this thesis is the Ptolemy II project [Pto14]. It is a tool for exploring model-based design and has a strong focus on CPSs. Ptolemy II is very comprehensive and studies and implements several MoCs discussed in this thesis (cf. Section 6.1). The scope of Ptolemy II is far larger and more detailed than this thesis. It is, however, aimed at application developers. In contrast, many methods in this thesis (chapters 4-5) are more focused on tool developers, for improving the methods enabling model-based design.

Instead of discussing related work generally, we will go over on the different methods proposed here to improve model-based design, and discuss related work by broader categories. In most chapters and sections we discuss related work directly. While we will systematically go over related work here, we refer to the according chapters and sections for discussion.

8.1 Dataflow-based Software Synthesis

There are many tools for software synthesis, as we discuss in Section 2.6. We have mentioned [Lin98] which uses Petri Nets, or [RPM92] based on SDFs. The flows in [BLM00; Pin+95; BML12] are generally based on dataflow.

More recently, there is SystemCoDesigner [Hau+08], which is based on SystemC and targets Field Programmable Gate Arrays (FPGAs). The same is the case for the dataflow-based CAPH [SBA13] framework. On the software side, there is the Turnus [Cas+13] flow, which is based on RVC-CAL. Also relevant is the PREESM [Pel+14] flow, which is based on parametrized extensions of SDF models. These flows are all related to MAPS [CLA11], and its spinoff in Silexica, which is the KPN-based software synthesis flow that we focus on in this thesis, and describe in Chapter 2. As such, the more closely related tools are KPN-based flows, like Sesame [Erb+07], with the related ESPAM [SDN06] and Daedalus [Nik+08]. Similarly, the DOL [Thi+07] is a closely related KPN-based flow.

While we did not propose a new software synthesis flow, the methods in this thesis and in `mocasin` are related to methods implemented in these diverse flows. To the best of our knowledge, there are no systematic comparisons of approaches and heuristics in terms of their performance, as we argue in [Goe+16]. The survey in [Sin+13] is a systematic comparison of mapping approaches at an abstract level, but it does not execute and compare the different heuristics in benchmarks. The work in [Bra+01], does execute and compare heuristics, on the other hand, but these are from before the multicore era and not as directly related.

8.2 Mapping Space Structures

In [TP13], Thompson and Pimentel exploit the mapping space structure explicitly, exploiting both symmetries of the problem and a kind of metric for the mapping space in the form of operators for genetic algorithms. These can both be seen as special cases of the structures in Chapter 4, albeit for a simpler case with homogeneous architectures. The work from Richthammer and others [RG18; RFG20] is very similar in nature to the applications discussed in Chapter 5. They also aim to improve DSE methods from an algorithm-agnostic fashion, although the concrete structure they exploit is different. Their methods are orthogonal to ours (and could be combined). Less directly related are approaches for pruning the design space in general settings, outside the mapping problem [WS04].

8.2.1 *Symmetries*

Symmetries have been explored in software synthesis implicitly in many cases, e.g. in [HT01; Kre+05; Sin+10; Rol+15]. In fact, when researchers or developers just distinguish between core types in architectures with simple memory subsystems they are implicitly considering the symmetries of the problem. The problem becomes more difficult when the architecture topologies are more complex. The authors of [Sch+17] also consider symmetries in DSE, albeit in a more ad-hoc fashion (without the mathematical theory of groups or semigroups). In a related idea, in [Wei+16] they also introduce the concept of “shapes”, which are a special case of the symmetries exploited in the TETRiS method, but is limited to meshes in NoCs. For some applications, the symmetries have also been considered explicitly [Coh88], but not systematically like we do in this thesis.

Methods from group theory have also been used to exploit symmetries for problems in computer science and engineering before, in ways that are very similar to the methods discussed in this thesis [Cra+96; Cla+98]. In particular, some of our methods are inspired by the usage of wreath products in model checking [DM09].

8.2.2 *Distances*

Distances between mappings are commonly described in NoC-based systems. For example, the heuristic described in [Sin+10] considers mappings in NoC systems and uses the number of hops in the topology to find them. This strategy is common in many approaches. In [Wei+14], for example, the authors encode a related notion of distance in the constraints of their operating points. To the best of our knowledge, explicit low-distortion embeddings have not been used in this context before.

Robustness in computation is a broad subject and much work has been done in different aspects of it, albeit most of it does not consider robust mappings explicitly. The work of [ZK11; Zha19] defines mapping migration strategies, without focusing on a DSE for robust mappings. A strategy for finding robust mappings explicitly was proposed in [Che+16], using redundancy instead of the geometry of the mapping space. This allows for more robustness but requires more resources. Design centering methods, like the ones we used to find robust mappings, have also been used in many other disciplines in engineering, e.g. integrated-circuit design [Che+15].

To the best of our knowledge, the only other work defining compactness of mappings explicitly is [Yan+10]. However, the quality of this work, including the heuristic and its evaluation, is dubious. The idea behind compactness of mappings is composability of applications, on the other hand, which has been seriously researched with other methods. CoMPSoC [Han+09] or the work in [Kum+08] deal with composability of applications. They do not do so using the geometry of the mappings[1], however, but using sophisticated hardware support instead.

8.3 Run-time and hybrid approaches

There are many run-time and hybrid flows related to our TETRiS approach. The flows proposed in [Cas+10; Kan+14; Zhu+16], for example, also propose methods for multi-application mappings. These methods all rely on statically knowing all applications at compile time and calculating joint mappings, which does not scale nor works in more dynamic systems.

The approaches from DAARM [Wei+14] or Spider [Heu+14], or the methods proposed in [MMB07; QP15] are all hybrid approaches. As such, they all solve the problem with static approaches discussed above, like TETRiS does. These rely on different methods for finding the final mappings at run-time and have different advantages. Most works have the architecture model implicit in the flow (cf. Section 2.3), e.g. [Wei+14] which assumes regular NoC meshes from its problem formulation or [MMB07]. A distinct advantage of our symmetries-based approach to hybrid mapping is that it uses a general architecture model, which works for arbitrarily complex architectures.

On the side of run-time adaptivity, the MacQueen gap is generally ignored in literature, where KPN are equated with the KMQ blocking-read semantics. We were not the first to recognize the gap, however. The two models are also treated as separate in [LM09]. There are many other related ways of adapting MoCs at run-time. Models like SADF [The+06] or Multi-Alternative Process Networks [BJC21] do this by modeling the adaptivity explicitly in the graph. The AdaPNet model [Sch+14] defines more comprehensive transformations and is very general and flexible.

On the side of applications to 5G, mostly the complete field of research in telecommunications investigates methods for adapting to the dynamic demands of upcoming applications. Models are common for different areas of the field, but their use is generally not proposed at a system-level as we do. An example that is proposed at a system-level is the Nucleus project [Cas+11], that is based on the MAPS software synthesis flow and KPNs. Since this is work-in-progress, it is not possible to provide a detailed examination of advantages and disadvantages of our proposed approach using Reactors, compared to established methods, as we have yet to fully examine and understand these. A good overview of model-based approaches in modem design can be found work in [Gat+20].

8.4 Other model-based design tools

The tools above focus on software synthesis in a flow similar to the one described in this thesis, using KPN or dataflow models and DSE to find profitable mappings for executing applications in modern hardware. There

---

[1] Which is not a good strategy, as we saw in Section 5.1

are other tools and languages which are focused more in the model's semantics. Besides Ptolemy II [Pto14], these include the CAL [EJ03] language and the related RVC-CAL compiler. Synchronous languages like LUSTRE [PHP87] or ESTEREL [BD91], which have discrete-event semantics, are also relevant. These are languages are related to the Reactors model.

Even hardware description languages like VHDL or Verilog, and even SystemC [Mue+01] can be seen as related, in this case it being HLS, which is in fact an inspiration for the term software synthesis. Also on the commercial side, Signal Processing Work System and Synopsys System Studio both from Synopsis join the LabVIEW Communications System Design Suite or Matlab Simulink [KKM16] as model-based design tools with well-defined MoCs. In Section 7.1.1 we review and discuss many of these tools and programming languages based on MoCs.

8.5 Random Benchmark Generation and Machine Learning

A prominent example of random code generation is CSmith [Yan+11], used to stress-test compilers. A related approach is the grammar-based method presented in [McK97]. In this thesis we discussed random benchmarking for evaluating optimizations, more so than for testing corner cases. The tools TGFF [DRW98] and SDF^3[SGB06], also discussed in Section 3.1 are more closely related to benchmarking as we investigated it.

We used the proposed Level graphs to evaluate language-based transformations for I/O optimization. We discussed the main related work for this in Section 7.3, namely Haxl [Mar+14] and Muse [Kac15], as well as the unpublished Stitch [Don14] from Twitter.

The direct connection between benchmarking and machine learning comes from CLGen [Cum+17a]. Machine learning for code is a broad subject and only a minor part of this thesis. A broader overview can be found in [All+18], although we will solely discuss work closely related to the contributions presented in this thesis. We based our graph-based methods and the evaluation on the ideas presented in [Cum+17b; BJH18]. Consequently, based in part on our graph-based methods, the work in [Cum+20; Ye+20] recently proposed some potential improvements to our compiler-based representations.

9

CONCLUSIONS

Programming computers is notoriously difficult. This thesis certainly will not change that. More generally, this statement will probably remain true for a long time. This does not mean, however, that we cannot make progress towards easing the process of programming computers.

In this thesis we have argued for model-based design of software systems. This is much more common in the hardware world, where models are central to design, commonly used for ensuring deterministic behavior. The success of this paradigm is what allows us to have programmable digital computers in the first place.

In the software world, where the level of abstraction is higher and the entry barrier lower, models are usually more implicit, less strict, or both. When using well-defined Models of Computation (MoCs) for programming, however, we can reason about software and its performance. Concretely, software synthesis flows and the mapping problem result from doing precisely this.

In this thesis we studied such MoC-based software synthesis flows, with a focus on Kahn Process Network (KPN). We surveyed multiple dataflow MoCs, and discussed the advantages and disadvantages of them. The KPN MoC allows us to express concurrency in computation in a deterministic fashion, while remaining very expressive. Compared to most dataflow MoCs, it allows for maximally dynamic, data-dependent behavior. We also discussed a semantics gap between the Kahn-MacQueen (KMQ) blocking-reads semantics and KPN, which can be exploited in applications with data-parallelism.

When lowering KPNs down to be executed in MPSoCs, the mapping problem plays a crucial role, especially for heterogeneous systems. In this thesis we have discussed this intractable problem at length. A central theme of our discussion has been the structure of the mapping space. We have seen how the space is large and complex, yet structured.

The mapping space is very symmetrical, which concretely means that many mappings are equivalent in terms of properties like performance or energy efficiency. This is due to symmetries in the target hardware architectures and applications. MPSoCs usually have multiple cores with identical microarchitectures and memory subsystems with a regular structure. Data-level parallelism in applications also yield such symmetry. We have seen how to describe and exploit this symmetry, pruning the mapping space for Design-Space Exploration (DSE) or for finding equivalent mappings when some resources are unavailable at run-time.

The mapping space also has different geometric interpretations. We have seen how to find different embeddings of these geometric interpretations and exploit them in DSE meta-heuristics. This also allowed us to design novel heuristics and meta-heuristics, based on the geometric structure of the space, to find mappings with low communication costs or high robustness. In general, we have seen how the way we represent mappings can expose much of this structure. We believe much work would benefit from explicitly considering the structure exposed to the algorithms.

There is little point in exposing complex structures and engineering sophisticated algorithms if they don't improve our methods. To assess if they do, however, we need to test them, using benchmarks. Given the importance of this, we argue for careful consideration as to what and how improvements are assessed using benchmarks. In this thesis we have argued for a statistical view of code, seeing improvements on methods as improvements on the expected value of some property, like the program's execution time.

Unfortunately, benchmarks are scarce and seldom specialized. We have discussed options for is overcoming this issue, using random benchmark generation and machine learning. In particular, we have seen how our statistical view of code for benchmarking exposes some possible pitfalls of machine learning for benchmark generation, both in theory and in practice.

While KPN-based flows have many advantages, they are not well-suited for every application domain. For example, KPNs do not have semantics to deal with time, which is important in Cyber-Physical Systems (CPSs). Similarly, the KPN graph structure is rigid, which limits the adaptability of the model. In this thesis we discussed a novel model, Reactors, which addresses these limitations. We focused on the opportunities of this model in the 5G telecommunications standard.

Most of this thesis has focused on the advantages of model-based design, which are plentiful. An important disadvantage, however, is the ease of use of this design process. Exposing models through APIs is not productive, since developers can and usually do end up abandoning the model's constraints. We need programming languages and, especially, programming models that make MoC-based design accessible to programmers, while enforcing the model's constraints. In this thesis we briefly discussed the Ohua programming model, which derives a dataflow execution implicitly from a conventional programming language. We saw how we can use this to combine the advantages of MoC-based design with concise programming, by optimizing I/O in microservice-based infrastructures.

9.1 Future Work

We believe the single most important aspect to drive MoC-based design forward is fostering its adoption. We need tools and environments that make it easier for programmers to design applications with a well specified MoC.

The Lingua Franca project, which implements Reactors, is a great avenue for fostering adoption. Its polyglot design allows programmers to use known languages to write reactors[1], while still being a coordination language that can enforce the MoC semantics. The use of known languages has two distinct advantages, as it reduces the learning curve and allows using legacy code. Currently, the compiler does not understand nor does it type-check the target language, leaving that task to the compiler. In future work we could add a type system to Lingua Franca, reinforcing the "freedom from choice" it provides.

A potential disadvantage from the Lingua Franca project, on the other hand, could be its coordination language. Explicitly writing the networks can be confusing for developers. This is exacerbated by the fact that Reactors is a complex model, difficult to grasp and learn. We believe the

---

[1] This refers to the unit of computation, reactors, as part of the model, Reactors.

approach by Ohua of making the network implicit is a great avenue for future work. Another example to follow is the Elm language, a functional language for the web. Elm started as a language with an explicit Functional Reactive Programming (FRP) paradigm, which was confusing for users. They made this paradigm implicit[2], which translated into a success for the learning curve and the language's adoption. This is part of a general vision, where compilers and languages are thought of as assistants that make development easier for programmers, instead of only focussing on correctness and performance. We believe we should follow similar paths for the design of software using MoCs like Reactors or KPN.

On the side of mapping there are also many open avenues for improving our work. In particular, partial symmetries expose a great deal of the problem's structure which we are not exploiting yet. Designing efficient methods to detect and exploit them would help navigate the design space of mappings much better. This would also open up opportunities for using inverse semigroups of non-symmetries, like reducing the number of hops in a communication link. Our methods would also greatly benefit from incorporating application symmetries when exploiting data-level parallelism.

The geometry of the mapping space also has many open questions that should improve its usefulness. While we discussed the trade-off between the distortion and dimensionality of embeddings, we did not exploit it in this thesis. More importantly, while the metrics we discussed are a good starting point, we saw that they do not reflect the mapping structure very well yet. Finding better metrics could greatly improve mapping meta-heuristics, especially those based on some concept of locality in the search space.

In this thesis we have discussed and shown how provable properties of MoCs can and do improve the design process. While traditional pen and paper proofs are a great way of finding and proving these properties, the advent of powerful theorem proving assistants provides an opportunity to improve upon this. Formally verified proofs of properties of the system give us more certainty on their correctness, and some degree of automation. Combining MoC-based design with developments in formal verification methods can help us to write correct software more frequently in less time.

While formal methods can verify properties of our software, they might never completely replace testing. Even if they one day do so, that will not be in the near future. This is why we believe the work on benchmarking is an important avenue for future work. We believe advances in machine learning could enable our vision of benchmark generation flows. Informed by statistical models and tailored for a specific use-cases, such benchmark generation flows could dramatically change the way we assess our compilers.

---

[2] `https://elm-lang.org/news/farewell-to-frp`

A

MATHEMATICAL SUPPLEMENT

A.1 Groups

Definition A.1.1. Let G be a (finite[1]) set and $\circ : G \times G \to G$ be a mapping. We say that $(G, \circ)$ is a *group* if the following hold:

- The mapping $\circ : G \times G$ is *associative*, i.e. for any $f, g, h \in G$ we have $f \circ (g \circ h) = (f \circ g) \circ h$.
- There exists a neutral element $e \in G$ with $g \circ e = e \circ g = g$ for all $g \in G$.
- For every $g \in G$ there is an *inverse* element $g^{-1} \in G$ such that $g \circ g^{-1} = g^{-1}g = e$.

By abuse of notation, we normally identify the structure $(G, \circ)$ with the set G and say G is a group. Similarly, when the operation $\circ$ is understood from context we commonly abbreviate it as multiplication, without writing it explicitly: $g \circ h =: gh$. Groups are ubiquitous in mathematics. For example, the natural numbers form a group with addition $(\mathbb{N}, +)$, as do the reals (without 0) with multiplication $(\mathbb{R} \backslash \{0\}, \cdot)$.

An important example of a group is the so-called *symmetric group*:

Example A.1.2. Let X be a finite set. Then, the set of bijections $X \to X$ from X to itself is a group with regard to function composition. Indeed, if $f, g : X \to X$ are bijections, then so is $f \circ g : X \to X$. The identity function $\mathrm{Id}_X : x \mapsto x$ is the neutral element and the inverse function f^{-1} is the group inverse of f, since $f \circ f^{-1} = f^{-1}f = \mathrm{Id}_X$. We call the group of bijections on X the symmetric group on X and write $\mathrm{Sym}(X)$. If $n \in \mathbb{N}, n > 0$ is a natural number and $X = \{1, \ldots, n\}$, then we write S_n to refer to $\mathrm{Sym}(\{1, \ldots, n\})$.

This is an important example because every finite group can be found in S_n for some n, as a subgroup, which we will define shortly. We first want to introduce cycle notation. A permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ can be written in different ways. The simplest way to do this is to write it in a two-row matrix:

$$\begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

While this is simple to understand, there is a more concise way to write permutations that has many advantages, including some computational advantages. It is called cycle notation. We write a permutation as a product of cycles $(i\ \pi(i)\ \pi(\pi(i))\ \ldots\ \pi^k(i))$, maximal such that the values don't repeat. We can do this since n is finite and thus for some k, $\pi^{k+1}(i) = i$, and we choose the minimal k with that property. For example, the cycle $(1, 2, 3)$ is the permutation $1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 1$. From this example it perhaps is clearer why they are called cycles, since the last element maps to the first one, cyclically. By convention, 1-cycles are not written

---

[1] Groups are not finite by definition, but all groups we discuss in this thesis are.

explicitly. The identity mapping can be sometimes be written as $()$, but could equivalently be $(1)(2)\dots(n)$. For another example, the permutation $1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3, 4 \mapsto 5, 5 \mapsto 4$ can be written as $(1,2)(3,4,5)$.

Definition A.1.3. Let $H \subseteq G$ be a subset of a group G. We say that H is a subgroup if H is a group with respect to the restriction of the multiplication on G. Equivalently, if $e \in H$ and for every $g, h \in G$ we have $gh^{-1} \in G$.

We normally write $H \leqslant G$ to denote that H is a subgroup of G (and $H < G$ if, additionally, we know that $G \neq H$).

In group theory in particular, and in mathematics in general, mappings that preserve the structures of the objects being studied are a very powerful tool. We proceed to define these mappings for groups.

Definition A.1.4. For two groups G, G', a mapping $\varphi : G \to G'$ is called a group homomorphism (or a morphism of groups) if it respects the group's structure, i.e. if $\varphi(gh) = \varphi(g)\varphi(h)$ for all $g, h \in G$ and $\varphi(e_G) = e'_G$.

The definition above already implies that $\varphi(g^{-1}) = \varphi(g)^{-1}$. As mentioned before, these structure-preserving mappings are very important in the theory of groups, as they are used to relate different mappings. A group homomorphism $\varphi : G \to H$ is called a *monomorphism* (or embedding) if it is injective, *epimorphism* if it is surjective and *isomorphism* if it is bijective. A group isomorphism from a group G to itself, $\varphi : G \to G$ is called an *automorphism*. Isomorphisms play a central role in mathematics (e.g. in their more general definition for categories). They define equivalence classes of objects, i.e. being isomorphic is an equivalence relation. We usually write $G \cong H$ to say that G and H are isomoprhic, that is to say, if there exists an isomoprhism $\varphi : G \to H$. Two objects that are isomorphic are usually consider to be "the same", since any structural property has to be an invariant of the isomorphism class. In fact, this indistinguishability between isomorphic objects is at the center of the univalance axiom in homotopy type theory as a foundation of mathematics [Uni13].

In the case of groups, there is a particular property that most other structures in mathematics do not have. The set of isomorphisms, i.e. the set of mappings that preserve the structure of an object and define equivalences between objects, is itself a group! Besides group automorphisms, the structure-preserving mappings of other structures are also groups (e.g. homeomoprhisms in topological spaces, graph isomorphisms, invertible matrices in vector spaces). In all these cases there is a direct relationship between the structure and the structure preserving mappings, as the mappings can transform these structures. This concept is generalized with group *actions*.

Definition A.1.5. Let G be a group and X be a set. We say that G acts on X if there is a group homomorphism $G \to \mathrm{Sym}(X)$ from the group G to the symmetric group on X. Equivalently, if there is an $\alpha : G \times X \to X$ which is associative and respects the group operation (in particular $\alpha(e, x) = x$ for all $x \in X$)[2]. We also say that X is a G-set.

As an example, consider a regular polygon with n sides, a regular n-gon. Figure A.1 shows this for the example of a square (a square is a regular 4-gon). We name the four corners of the square as $1, 2, 3, 4$. This could thus be interpreted as a graph $G = (V = \{1,2,3,4\}, E = \{\{1,2\},\{2,3\},\{3,4\},\{4,1\}\})$.

---

[2] We call this a *left* action, and a right action similarly $\beta : X \times G \to X$, where we write the action of the group on the set from the right.

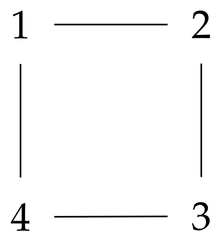


Figure A.1: A square.

The group of permutations S_4 acts on the graph G by permuting the points $V = \{1, 2, 3, 4\}$ (and the edges accordingly). We can take any permutation $\pi \in S_4$ and apply it on the graph. For example, consider the permutation $(1, 2)$ which swaps the two nodes 1 and 2 and leaves the rest as is.

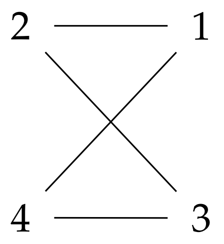


Figure A.2: The action of the permutation $(1, 2)$ on the square.

Figure A.2 shows the example of the action of $(1, 2)$ on the square. We note that the square is not a square anymore, it has lost its structure. A natural question to ask is, what are the permutations that leave a square as a square, i.e. preserve its structure? We call these the *symmetries* of the square .

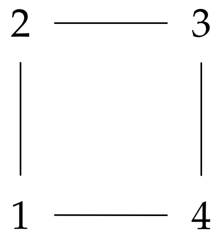


Figure A.3: The action of the rotation $(1, 2, 3, 4)$ on the square.

Consider a rotation by 90°, counter-clockwise. We can write this as the permutation $\rho = (1, 2, 3, 4)$. Figure A.3 depicts the action of ρ on the square, and indeed, it preserves the structure. We can think of two additional rotations, by 180° and 270°, which would also preserve the structure of the square. It is worth noting that a rotation by 180° is the same as rotating by 90° twice, and similarly, ρ^3 is the rotation by 270°. It is also important to note that a rotation by 360° and 0° are indistinguishable on the square, they are the identity permutation $\mathrm{Id}_{\{1,2,3,4\}}$. In fact, these four rotations form a sub-group $C_4 < S_4$, called a cyclic group. More generally, the rotations that preserve the structure of a regular n-gon are a cyclic group of order n, C_n.

Definition A.1.6. The cyclic group C_n is the group formed by an n-cycle $a = (1, \dots, n)$ in S_n, i.e. $C_n = \{a, a^2, \dots, a^n = \mathrm{Id}_n\}$.

We have seen that the rotation ρ in the example above is enough to find all elements of the group C_4. We say that ρ *generates* the group C_4. Cyclic groups are characterized by having a single generator [3]. Thus, all other groups have multiple generators (except the trivial group $\{e\}$ which can be considered as having 0 generators.) More generally, for a set $X \subseteq G$ in a group G, we define $\langle X \rangle \leqslant G$ to be the smallest subgroup of G containing X. For the case of finite groups, we can characterize $\langle X \rangle$ as the set of words in G (where we interpret concatenation of words as multiplication in the group).

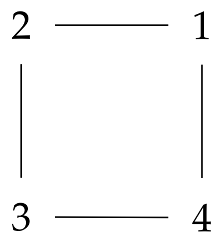


Figure A.4: The action of the reflection $(1,2)(3,4)$ on the square.

Rotations are not all the symmetries of the square, we can also have reflections. Figure A.4 shows the action of a reflection along the vertical axis, namely $\sigma = (1,2)(3,4)$, on the square. Reflections are fundamentally different from rotations, no rotation could achieve this transformation. We can also have a reflection along the horizontal axis and both diagonals, for a total of 4 reflections. It is perhaps not obvious at first, but if we combine reflections and rotations on the square, we always get a reflection or a rotation. In fact, these 8 transformations form another group D_4, with $C_4 < D_4 < S_4$. The dihedral group on four points, D_4, in generated by the rotation ρ and the reflection σ, i.e. $D_4 = \langle \rho, \sigma \rangle$.

In the action of permutations on the graph, the group elements act simultaneously on both the nodes and edges. If we look at the action only on the nodes, the action of C_4 (and D_4) can take any point to any other point. But if we look at the whole graph, we see that there is no way to take the original square to the square in Figure A.4 with the action of C_4. Similarly, we cannot use any group element of D_4 to take the shape in Figure A.2, which is not a square anymore. This is precisely the property that defines D_4 as the symmetry group of the square. These questions, concerning which elements can be taken which others, are common in group theory. This is why there is a definition to describe these sets: we call them orbits .

Definition A.1.7. Let G be a group and X be a G-set. Further, let $x \in X$ be an element of X. We define the *orbit* of x to be the set $Gx := \{gx \mid g \in G\}$ of points that x can be transfromed into. We further call $X/G := \{Gx \mid x \in X\}$ the set of orbits of X.

---

[3] Recall that we are only considering finite groups.

For example, all squares are precisely the elements in the orbit of $D_4 s$, where s is the graph of the square from the examples above. Orbits define a partition on the set X, meaning that any two orbits Gx, Gy are either equal or disjoint and $X = \bigcup_{x \in X} Gx$.

Finally, we discuss how we can construct other groups from existing groups. The simplest construction is called the direct product. For groups G, H, we write $G \times H$, endowing the Cartesian product with a component-wise multiplication (i.e. $(g,h)(g',h') = (gg',hh')$ for all $g,g' \in G, h,h' \in H$). There is a more general construction called a semi-direct product, which is a generalization of the direct product. Here we will only discuss a special case of semi-direct products, namely the *wreath product* $G \wr H$.

Let G be a group and let $H \leqslant S_n$ for an $n \in \mathcal{N}$. We consider the direct product of n copies of G:

$$G^n := G \underbrace{\times \ldots \times}_{n \text{ times}} G$$

Then, the group H acts on these n copies of G by permuting their instances. Let $(g_1, \ldots, g_n) \in G^n, h \in H$. We define:

$$^{h}(g_1 \ldots, g_n) := (g_{h1}, \ldots, g_{hn}),$$

where h permutes the order of the elements in the n-tuple of elements of G^n. This defines an action of H on G^n. We can use this action to construct the wreath product $G \wr H$ on the Cartesian product $G^n \times H$, by defining the multiplication as:

$$\begin{aligned} &((g_1, \ldots, g_n), h)((g'_1, \ldots, g'_n), h') \\ &= ((g_1, \ldots, g_n)\, {}^{h}(g'_1, \ldots, g'_n), hh') \\ &= ((g_1, \ldots, g_n)(g'_{h1}, \ldots, g'_{hn}), hh') \end{aligned}$$

Intuitively, the wreath product works when we have copies of a substructure arranged in a particular larger structure. It applies transformations both at the substructure level and an the level of the larger structure.

a.2 Metric Spaces and Low-Distortion Embeddings

Here we discuss (discrete) metric spaces and low-distortion embeddings. A metric space is the mathematical formalization of distances. We define a metric to be able to measure distances in a particular space.

Definition A.2.1. Let M be a set and let $d : M \times M \to \mathbb{R}_{\geqslant 0}$. We say that d is a metric on M and, equivalently, (M, d) is a metric space, if the following hold:

1. For all $m, m' \in M, d(m, m') = 0 \Leftrightarrow m = m'$.
2. For all $m, m' \in M$, we have $d(m, m') = d(m', m)$.
3. For all $k, l, m \in M$ we have $d(k, m) \leqslant d(k, l) + d(l, m)$

The motivation for these properties is intuitively clear. Property 1 says two things, first, that there is no distance from an element to itself, and second, that no two equal elements are in the same place (have no distance between them). If we don't require the second property (i.e. replace

$\Leftrightarrow$ with $\Rightarrow$ in Property 1, we get what is called a pseudo-metric (or a degenerate metric). The second property, Property 2 states that distance is symmetric. Finally, Property 3 is the triangle inequality: it states that the shortest path between two elements is always the direct path, their distance.

The canonical metric spaces are $\mathbb{R}^n$ with different norms, like the p-norms. A norm is a more restrictive concept than a metric, but we will not define norms here further.

Example A.2.2. For $p \geqslant 1$, the function $(x, y) \mapsto \|x - y\|_p : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_{\geqslant 0}$ is a metric, where $\|(x_1, \dots, x_n)\|_p := (\sum_{i=1}^n |x_i^p|)^{1/p}$ is the p-norm.

The case for $p = 2$ is the well-known Euclidean distance in vector spaces. Also well-known is the case of of $p = 1$, which is sometimes called the Mahattan or Taxi distance, in allusion to the distance when moving through the streets of a neighborhood that look like a regular mesh, like in Manhattan.

In this thesis we are particularly interested in the case where M is finite, which we will assume from here on. If $M = \{m_1, \dots, m_n\}$ is finite, we can write d as a matrix, such that $d(m_i, m_j) = D_{i,j}$.:

$$D = \begin{pmatrix} d(m_1, m_1) & d(m_1, m_2) & \dots & d(m_1, m_n) \\ d(m_2, m_1) & d(m_2, m_2) & \dots & d(m_2, m_n) \\ \vdots & \vdots & \ddots & \vdots \\ d(m_n, m_1) & d(m_n, m_2) & \dots & d(m_n, m_n) \end{pmatrix}$$

The structure preserving mappings (moprhisms) of metric spaces are called *isometries*. They have the particular property that they are always injective, due to Property 1.

Definition A.2.3. Let M, M' be metric spaces. We say that a mapping $\varphi : M \to M'$ is an *isometry* if for all $m, m' \in M$, $d_M(m, m') = d_{M'}(\varphi(m), \varphi(m'))$.

An isometry is thus always an embedding (monic), since for any two points $m, m' \in M$ with $\varphi(m) = \varphi(m')$ we have $0 = d_{M'}(\varphi(m), \varphi(m')) = d_M(m, m')$.

In the case of groups, embeddings into a particular group S_n are useful for computing. While we did not discuss it as thoroughly, the basis of all computation we are concerned with in this thesis are these embeddings into permutation groups S_n[4]. The question is, can we find an equivalent method for metric spaces, using isometries to (finite subsets of) $\mathbb{R}^n$? The unfortunate answer is that no, for a finite metric space M there is not always n, p such that there exists an isometry from M to $(\mathbb{R}^n, \| \cdot \|_p)$ (see [Mat02] for a proof). Fortunately, however, when dealing with real numbers we can always look for approximations.

Definition A.2.4. Let M be a metric space and $\iota : M \hookrightarrow \mathbb{R}^n$ be an embedding onto $\mathbb{R}^n$. We say that ι has distortion $D > 0$ if

$$\frac{1}{D} d(x, y) \leqslant \|\iota(x) - \iota(y)\| \leqslant d(x, y)$$

---

[4] In computational group theory there are other branches like matrix groups or black-box groups, where embedding into an S_n is infeasible, but we are not concerned with these in this thesis.

While, in general, we cannot find an isometry, we can search for an embedding with a low distortion. There is a particularly useful result in this context: we can use convex optimization to find an embedding of M onto $\mathbb{R}^n$ with the Euclidean ($p = 2$) norm [Mat02]. This is unfortunately only the case for this norm, e.g. for $p = 1$ finding such an embedding is known to be NP-complete [Mat02]

A problem with the convex optimization method above is that it yields an embedding with dimension $|M|$, which might be very high. The dimension of the vector space strongly affects algorithmic properties of the problem. It would be ideal to find an embedding into a mapping with a lower dimension, without increasing the distortion much. In [JL84], Johnson and Lindenstrauss describe this precise problem and its solution as follows: "Given n points in Euclidean space, what is the smallest $k = k(n)$ so that these points can be moved into k-dimensional Euclidean space via a transformation that expands or contracts all paairwise distances by a factor of at most $1 + \epsilon$? The answer, that $k \leqslant c(\epsilon) \operatorname{Log} n$, is a simple consequence of the isoperimetric inequality for the n-sphere studied in [FLM77]." They proceed to formalize and prove this fact, which we will not restate here more precisely. This result is known as the Johnson-Lindenstrauss Lemma.

An intuitive albeit sometimes misleading interpretation of the proof of this lemma is that a projection onto a **random** subspace will have a low distortion with high probability. In practice, the distribution does give a very useful "transform" for dimensionality reduction, simply by projecting onto a random subspace. However, we should be careful when using this and ideally check the distortion, if possible.

[Chu36] Alonzo Church. “An unsolvable problem of elementary number theory.” In: *American journal of mathematics* 58.2 (1936), pp. 345–363.

[Kle36] Stephen Cole Kleene. “General recursive functions of natural numbers.” In: *Mathematische annalen* 112.1 (1936), pp. 727–742.

[Tur37] Alan M Turing. “Computability and λ-definability.” In: *The Journal of Symbolic Logic* 2.4 (1937), pp. 153–163.

[ER59] P Erdős and A Réyni. “On random graphs I.” In: *Publicationes Mathematicae* 6.290-297 (1959), p. 18.

[Gil59] Edgar N Gilbert. “Random graphs.” In: *The Annals of Mathematical Statistics* 30.4 (1959), pp. 1141–1144.

[Pet62] Carl Adam Petri. “Kommunikation mit automaten.” In: (1962).

[Moo+65] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[Sco70] Dana Scott. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

[Kar72] Richard M Karp. “Reducibility among combinatorial problems.” In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[HBS73] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence.” In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*. 1973, pp. 235–245.

[Den74] Jack B Dennis. “First version of a data flow procedure language.” In: *Programming Symposium*. Springer. 1974, pp. 362–376.

[Kah74] Gilles Kahn. “The semantics of a simple language for parallel programming.” In: *Information processing* 74 (1974), pp. 471–475.

[KM76] Gilles Kahn and David MacQueen. *Coroutines and Networks of Parallel Processes*. Research Report. 1976, p. 20. url: `https://hal.inria.fr/inria-00306565`.

[FLM77] Tadeusz Figiel, Joram Lindenstrauss, and Vitali D Milman. “The dimension of almost spherical sections of convex bodies.” In: *Acta Mathematica* 139.1 (1977), pp. 53–94.

[Lam78] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*. 1978.

[JL84] William B Johnson and Joram Lindenstrauss. “Extensions of Lipschitz mappings into a Hilbert space.” In: *Contemporary mathematics* 26.189-206 (1984), p. 1.

[HP85] David Harel and Amir Pnueli. "On the development of reactive systems." In: *Logics and models of concurrent systems*. Springer, 1985, pp. 477–498.

[Hue85] Gérard Huet. "Cartesian closed categories and lambda-calculus." In: *LITP Spring School on Theoretical Computer Science*. Springer. 1985, pp. 123–135.

[Agh86] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. Cambridge, MA: MIT Press, 1986.

[Den86] Jack B Dennis. "Data flow computation." In: *Control Flow and Data Flow: concepts of distributed programming*. Springer, 1986, pp. 345–398.

[FW86] Philip J Fleming and John J Wallace. "How not to lie with statistics: the correct way to summarize benchmark results." In: *Communications of the ACM* 29.3 (1986), pp. 218–221.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors." In: *nature* 323.6088 (1986), pp. 533–536.

[LM87] Edward A Lee and David G Messerschmitt. "Synchronous data flow." In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.

[PHP87] Daniel Pilaud, N Halbwachs, and JA Plaice. "LUSTRE: A declarative language for programming synchronous systems." In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*. Vol. 178. 1987, p. 188.

[BB88] Jonathan Barzilai and Jonathan M Borwein. "Two-point step size gradient methods." In: *IMA journal of numerical analysis* 8.1 (1988), pp. 141–148.

[Coh88] Harvey A Cohen. "Symmetry considerations applied to hardware convolvers for image filtering." In: *Proceedings of the 1988 IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 2. IEEE. 1988, pp. 1128–1131.

[Run89] Colin Runciman. "What about the natural numbers?" In: *Computer Languages* 14.3 (1989), pp. 181–191.

[BD91] Frédéric Boussinot and Robert De Simone. "The ESTEREL language." In: *Proceedings of the IEEE* 79.9 (1991), pp. 1293–1304.

[MMP91] Oded Maler, Zohar Manna, and Amir Pnueli. "Prom timed to hybrid systems." In: *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer. 1991, pp. 447–484.

[Gun92] Carl A Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

[RPM92] Sebastian Ritz, Matthias Pankert, and Heinrich Meyr. "High level software synthesis for signal processing systems." In: *Proceedings of the international conference on application specific array processors*. IEEE Computer Society. 1992, pp. 679–680.

[Abb+93] Ben Abbott, Ted Bapty, Csaba Biegl, Gabor Karsai, and Janos Sztipanovits. “Model-based software synthesis.” In: *IEEE Software* 10.3 (1993), pp. 42–52.

[DR95] Volker Diekert and Grzegorz Rozenberg. *The book of traces*. World scientific, 1995.

[LP95] Edward A Lee and Thomas M Parks. “Dataflow process networks.” In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801.

[Maz95] Antoni W Mazurkiewicz. *Introduction to Trace Theory.* 1995.

[Par95] Thomas M. Parks. “Bounded Scheduling of Process Networks.” PhD thesis. EECS Department, University of California, Berkeley, Dec. 1995. url: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1995/2926.html`.

[Pin+95] José Luis Pino, Soonhoi Ha, Edward A Lee, and Joseph T Buck. “Software synthesis for DSP using Ptolemy.” In: *Journal of VLSI signal processing systems for signal, image and video technology* 9.1-2 (1995), pp. 7–21.

[Bil+96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. “Cycle-static dataflow.” In: *IEEE Transactions on signal processing* 44.2 (1996), pp. 397–408.

[Cra+96] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. “Symmetry-breaking predicates for search problems.” In: *KR* 96.1996 (1996), pp. 148–159.

[Nag+96] Wolfgang E Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. “VAMPIR: Visualization and analysis of MPI resources.” In: (1996).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory.” In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[McK97] Bruce McKenzie. “Generating strings at random from a context free grammar.” In: (1997).

[CDT98] G Calafiore, F Dabbene, and R Tempo. “Uniform sample generation in l/sub p/balls for probabilistic robustness analysis.” In: *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No. 98CH36171)*. Vol. 3. IEEE. 1998, pp. 3335–3340.

[Cla+98] Edmund M Clarke, E Allen Emerson, Somesh Jha, and A Prasad Sistla. “Symmetry reductions in model checking.” In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 147–158.

[DRW98] Robert P Dick, David L Rhodes, and Wayne Wolf. “TGFF: task graphs for free.” In: *Proceedings of the Sixth International Workshop on Hardware/Software Codesign.(CODES/CASHE’98)*. IEEE. 1998, pp. 97–101.

[Law98] Mark V Lawson. *Inverse semigroups: the theory of partial symmetries*. World Scientific, 1998.

[Lin98] Bill Lin. “Software synthesis of process-based concurrent programs.” In: *Proceedings of the 35th annual Design Automation Conference*. 1998, pp. 502–505.

[BLM00] SS Bhartacharyya, Ranier Leupers, and Peter Marwedel. “Software synthesis and code generation for signal processing systems.” In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47.9 (2000), pp. 849–875.

[Koc+00] Erwin A de Kock, WJM Smits, Pieter van der Wolf, J-Y Brunel, WM Kruijtzer, Paul Lieverse, Kees A Vissers, and Gerben Essink. “YAPI: Application modeling for signal processing systems.” In: *Proceedings of the 37th Annual Design Automation Conference*. 2000, pp. 402–405.

[Bra+01] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, et al. “A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems.” In: *Journal of Parallel and Distributed computing* 61.6 (2001), pp. 810–837.

[HT01] Frank Hannig and Jürgen Teich. “Design space exploration for massively parallel processor arrays.” In: *International Conference on Parallel Computing Technologies*. Springer. 2001, pp. 51–65.

[Kie+01] Bart Kienhuis, Ed F Deprettere, Pieter Van der Wolf, and Kees Vissers. “A methodology to design programmable embedded systems.” In: *International Workshop on Embedded Computer Systems*. Springer. 2001, pp. 18–37.

[Mue+01] Wolfgang Mueller, Juergen Ruf, Dirk Hoffmann, Joachim Gerlach, Thomas Kropf, and Wolfgang Rosenstiehl. “The simulation semantics of SystemC.” In: *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*. IEEE. 2001, pp. 64–70.

[Mat02] Jiří Matoušek. *Lectures on discrete geometry*. Vol. 212. Springer Science & Business Media, 2002.

[EJ03] Johan Eker and J Janneck. *CAL language report: Specification of the CAL actor language*. 2003.

[Ser03] Ákos Seress. *Permutation group algorithms*. Vol. 152. Cambridge University Press, 2003.

[SD03] Todor Stefanov and Ed Deprettere. “Deriving process networks from weakly dynamic applications in system-level design.” In: *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 2003, pp. 90–96.

[Sir04] Marjan Sirjani. “Formal specification and verification of concurrent and reactive systems.” In: *PhD thesis* (2004).

[WS04] G Gary Wang and Songqing Shan. “Design space reduction for multi-objective optimization and robust design optimization problems.” In: *SAE transactions* (2004), pp. 101–110.

[AGL05] James Ahrens, Berk Geveci, and Charles Law. “Paraview: An end-user tool for large data visualization.” In: *The visualization handbook* 717.8 (2005).

[Hol05] Derek F. Holt. *Handbook of Computational Group Theory*. CRC Press, 2005.

[Kre+05] Marcio Kreutz, César A Marcon, Luigi Carro, Flavio Wagner, and Altamiro A Susin. "Design space exploration comparing homogeneous and heterogeneous network-on-chip architectures." In: *Proceedings of the 18th annual symposium on Integrated circuits and system design*. 2005, pp. 190–195.

[ECP06] Cagkan Erbas, Selin Cerav-Erbas, and Andy D Pimentel. "Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design." In: *IEEE Transactions on Evolutionary Computation* 10.3 (2006), pp. 358–374.

[Kan+06] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. "UML-based multiprocessor SoC design framework." In: *ACM Transactions on Embedded Computing Systems (TECS)* 5.2 (2006), pp. 281–320.

[Lee06] Edward A Lee. "The problem with threads." In: *Computer* 39.5 (2006), pp. 33–42.

[PEP06] Andy D Pimentel, Cagkan Erbas, and Simon Polstra. "A systematic approach to exploring embedded system architectures at multiple abstraction levels." In: *IEEE Transactions on Computers* 55.2 (2006), pp. 99–112.

[SDN06] Todor Stefanov, Ed Deprettere, and Hristo Nikolov. "Multiprocessor system design with ESPAM." In: *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS'06)*. IEEE. 2006, pp. 211–216.

[SGB06] S. Stuijk, M.C.W. Geilen, and T. Basten. "SDF^3: SDF For Free." In: *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. Turku, Finland: IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006, pp. 276–278. doi: 10.1109/ACSD.2006.23. url: http://www.es.ele.tue.nl/sdf3.

[The+06] Bart D Theelen, Marc CW Geilen, Twan Basten, Jeroen PM Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. "A scenario-aware data flow model for combined long-run average and worst-case performance analysis." In: *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings.* IEEE. 2006, pp. 185–194.

[DeC+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store." In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[EL07] Stephen A Edwards and Edward A Lee. "The case for the precision timed (PRET) machine." In: *Proceedings of the 44th annual Design Automation Conference*. 2007, pp. 264–265.

[Erb+07] Cagkan Erbas, Andy D Pimentel, Mark Thompson, and Simon Polstra. "A framework for system-level modeling and simulation of embedded systems architectures." In: *EURASIP Journal on Embedded Systems* 2007 (2007), pp. 1–11.

[MMB07] Orlando Moreira, Jacob Jan-David Mol, and Marco Bekooij. "Online resource management in a multiprocessor with a network-on-chip." In: *Proceedings of the 2007 ACM symposium on Applied computing*. 2007, pp. 1557–1564.

[Ors+07] Heikki Orsila, Tero Kangas, Erno Salminen, Timo D. Hämäläinen, and Marko Hännikäinen. "Automated memory-aware application distribution for multi-processor system-on-chips." In: *J. of Sys. Arch.* 53.11 (2007), pp. 795–815.

[San07] Alberto Sangiovanni-Vincentelli. "Quo vadis, SLD? Reasoning about the trends and challenges of system level design." In: *Proceedings of the IEEE* 95.3 (2007), pp. 467–506.

[Thi+07] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. "Mapping applications to tiled multiprocessor embedded systems." In: *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*. IEEE. 2007, pp. 29–40.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[Dic08] Robert Dick. *Embedded Systems Synthesis Benchmark Suite (e3s)*. 2008. url: `http://ziyang.eecs.umich.edu/%5C~%7B%7Ddickrp/e3s/`.

[Hau+08] Christian Haubelt, Thomas Schlichter, Joachim Keinert, and Mike Meredith. "SystemCoDesigner: automatic design space exploration and rapid prototyping from behavioral models." In: *Proceedings of the 45th annual Design Automation Conference*. 2008, pp. 580–585.

[Kum+08] Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal, and Yajun Ha. "Analyzing composability of applications on MPSoC platforms." In: *Journal of Systems Architecture* 54.3-4 (2008), pp. 369–383.

[MH08] Laurens van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE." In: *Journal of machine learning research* 9.Nov (2008), pp. 2579–2605.

[MEP08] Sorin Manolache, Petru Eles, and Zebo Peng. "Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints." In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.2 (2008), pp. 1–35.

[Nik+08] Hristo Nikolov, Mark Thompson, Todor Stefanov, Andy Pimentel, Simon Polstra, Raj Bose, Claudiu Zissulescu, and Ed Deprettere. "Daedalus: toward composable multimedia MP-SoC design." In: *Proceedings of the 45th annual Design Automation Conference*. 2008, pp. 574–579.

[DM09] Alastair F Donaldson and Alice Miller. "On the constructive orbit problem." In: *Annals of mathematics and artificial intelligence* 57.1 (2009), pp. 1–35.

[EMD09] Wolfgang Ecker, Wolfgang Müller, and Rainer Dömer. "Hardware-dependent software." In: *Hardware-dependent Software*. Springer, 2009, pp. 1–13.

[HPP09] Mary Hall, David Padua, and Keshav Pingali. "Compiler research: the next 50 years." In: *Communications of the ACM* 52.2 (2009), pp. 60–67.

[Han+09] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. "CoMPSoC: A template for composable and predictable multi-processor system on chips." In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14.1 (2009), pp. 1–24.

[Lee09] Edward A Lee. "Computing needs time." In: *Communications of the ACM* 52.5 (2009), pp. 70–79.

[LM09] Edward A Lee and Eleftherios Matsikoudis. "The semantics of dataflow with firing." In: *G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, editors, From Semantics to Computer Science: Essays in Honour of Gilles Kahn* (2009), pp. 71–94.

[Cas+10] Jeronimo Castrillon, Ricardo Velasquez, Anastasia Stulova, Weihua Sheng, Jianjiang Ceng, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. "Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms." In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE. 2010, pp. 753–758.

[Sin+10] Amit Kumar Singh, Thambipillai Srikanthan, Akash Kumar, and Wu Jigang. "Communication-aware heuristics for runtime task mapping on NoC-based MPSoC platforms." In: *Journal of Systems Architecture* 56.7 (2010), pp. 242–255.

[SGB10] Sander Stuijk, Marc Geilen, and Twan Basten. "A predictable multiprocessor design flow for streaming applications with dynamic behaviour." In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE. 2010, pp. 548–555.

[Yan+10] Bo Yang, Liang Guang, Thomas Canhao Xu, Tero Säntti, and Juna Plosila. "Multi-application mapping algorithm for network-on-chip platforms." In: *2010 IEEE 26-th Convention of Electrical and Electronics Engineers in Israel*. IEEE. 2010, pp. 000540–000544.

[Bha+11] Shuvra S Bhattacharyya, Johan Eker, Jörn W Janneck, Christophe Lucarz, Marco Mattavelli, and Mickaël Raulet. "Overview of the MPEG reconfigurable video coding framework." In: *Journal of Signal Processing Systems* 63.2 (2011), pp. 251–263.

[CLA11] Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs." In: *IEEE Transactions on Industrial Informatics* 9.1 (2011), pp. 527–545.

[Cas+11] Jeronimo Castrillon, Stefan Schürmans, Anastasia Stulova, Weihua Sheng, Torsten Kempf, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. "Component-based waveform development: the Nucleus tool flow for efficient and portable software defined radio." In: *Analog Integrated Circuits and Signal Processing* 69.2 (2011), pp. 173–190.

[CSL11] Jeronimo Castrillon, Weihua Sheng, and Rainer Leupers. "Trends in embedded software synthesis." In: *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE. 2011, pp. 347–354.

[Mar+11] Peter Marwedel, Iuliana Bacivarov, Chanhee Lee, Jürgen Teich, Lothar Thiele, Qiang Xu, Georgia Kouveli, Soonhoi Ha, and Lin Huang. "Mapping of applications to MPSoCs." In: *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2011, pp. 109–118.

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers." In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 283–294.

[ZK11] Xiao Zhang and Hans G Kerkhoff. "A dependability solution for homogeneous MPSoCs." In: *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*. IEEE. 2011, pp. 53–62.

[BML12] Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. *Software synthesis from dataflow graphs*. Vol. 360. Springer Science & Business Media, 2012.

[Cas+12] Jeronimo Castrillon, Andreas Tretter, Rainer Leupers, and Gerd Ascheid. "Communication-aware mapping of KPN applications onto heterogeneous MPSoCs." In: *DAC Design Automation Conference 2012*. IEEE. 2012, pp. 1262–1267.

[Cza12] Evan Czaplicki. "Elm: Concurrent FRP for Functional GUIs." In: *Senior thesis, Harvard University* 30 (2012).

[Gaj+12] Daniel D Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification language and methodology*. Springer Science & Business Media, 2012.

[Cas+13] Simone Casale-Brunet, Claudio Alberti, Marco Mattavelli, and Jorn W Janneck. "Turnus: a unified dataflow design space exploration framework for heterogeneous parallel systems." In: *2013 Conference on Design and Architectures for Signal and Image Processing*. IEEE. 2013, pp. 47–54.

[Des+13] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine Aridhi. "Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration." In: *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE. 2013, pp. 41–48.

[OWG13] Michael FP O'Boyle, Zheng Wang, and Dominik Grewe. "Portable mapping of data parallel programs to OpenCL for heterogeneous systems." In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society. 2013, pp. 1–10.

[Ode+13] Maximilian Odendahl, Jeronimo Castrillon, Vitaliy Volevach, Rainer Leupers, and Gerd Ascheid. "Split-cost communication model for improved MPSoC application mapping." In: *2013 International Symposium on System on Chip (SoC)*. IEEE. 2013, pp. 1–8.

[SBA13] Jocelyn Sérot, François Berry, and Sameer Ahmed. “CAPH: a language for implementing stream-processing applications on FPGAs.” In: *Embedded Systems Design with FPGAs*. Springer, 2013, pp. 201–224.

[Sin+13] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. “Mapping on multi/many-core systems: survey of current and emerging trends.” In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2013, pp. 1–10.

[TDJ13] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. “Why do scala developers mix the actor model with other concurrency models?” In: *European Conference on Object-Oriented Programming*. Springer. 2013, pp. 302–326.

[TP13] Mark Thompson and Andy D Pimentel. “Exploiting domain knowledge in system-level MPSoC design space exploration.” In: *Journal of Systems Architecture* 59.7 (2013), pp. 351–360.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013.

[Vap13] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[Zhe+13] Qi Zheng, Yajing Chen, Ronald Dreslinski, Chaitali Chakrabarti, Achilleas Anastasopoulos, Scott Mahlke, and Trevor Mudge. “WiBench: An open source kernel suite for benchmarking wireless systems.” In: *2013 IEEE international symposium on workload characterization (IISWC)*. IEEE. 2013, pp. 123–132.

[CL14] Jerónimo Castrillón Mazo and Rainer Leupers. “Programming heterogeneous mpsocs: Tool flows to close the software productivity gap.” In: (2014).

[Don14] Jake Donham. *Introducing Stitch*. Tech. rep. [Online; accessed 4-May-2017]. 2014. url: `https://www.youtube.com/watch?v=VVpmMfT8aYw`.

[Eus+14] Juan Fernando Eusse, Christopher Williams, Luis Gabriel Murillo, Rainer Leupers, and Gerd Ascheid. “Pre-architectural performance estimation for ASIP design based on abstract processor models.” In: *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*. IEEE. 2014, pp. 133–140.

[Heu+14] Julien Heulot, Maxime Pelcat, Karol Desnos, Jean-Francois Nezan, and Slaheddine Aridhi. “Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps.” In: *2014 6th European Embedded Design in Education and Research Conference (EDERC)*. IEEE. 2014, pp. 167–171.

[Kan+14] Shin-haeng Kang, Hoeseok Yang, Sungchan Kim, Iuliana Bacivarov, Soonhoi Ha, and Lothar Thiele. “Static mapping of mixed-critical applications for fault-tolerant MPSoCs.” In: *Proceedings of the 51st annual design automation conference*. 2014, pp. 1–6.

[Mar+14] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. "There is no fork: An abstraction for efficient, concurrent, and concise data access." In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 2014, pp. 325–337.

[MP14] Brendan D. McKay and Adolfo Piperno. "Practical graph isomorphism, {II}." In: *Journal of Symbolic Computation* 60.0 (2014), pp. 94–112. issn: 0747-7171. doi: `http://doi.org/10.1016/j.jsc.2013.09.003`. url: `http://www.sciencedirect.com/science/article/pii/S0747717113001193`.

[Mur+14] Luis Gabriel Murillo, Simon Wawroschek, Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. "Automatic detection of concurrency bugs through event ordering constraints." In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–6.

[Noe+14] Benedikt Noethen, Oliver Arnold, Esther Perez Adeva, Tobias Seifert, Erik Fischer, Steffen Kunze, Emil Matúš, Gerhard Fettweis, Holger Eisenreich, Georg Ellguth, et al. "10.7 A 105GOPS 36mm 2 heterogeneous SDR MPSoC with energy-aware dynamic scheduling and iterative detection-decoding for 4G in 65nm CMOS." In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE. 2014, pp. 188–189.

[Pel+14] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming." In: *2014 6th european embedded design in education and research conference (EDERC)*. IEEE. 2014, pp. 36–40.

[Pto14] Claudius Ptolemaeus, ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. url: `http://ptolemy.org/books/Systems`.

[QP14] Wei Quan and Andy D Pimentel. "Towards exploring vast mpsoc mapping design spaces using a bias-elitist evolutionary approach." In: *2014 17th Euromicro Conference on Digital System Design*. IEEE. 2014, pp. 655–658.

[Sch+14] Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. "AdaPNet: Adapting process networks in response to resource variations." In: *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. 2014, pp. 1–10.

[She+14] Weihua Sheng, Stefan Schürmans, Maximilian Odendahl, Mark Bertsch, Vitaliy Volevach, Rainer Leupers, and Gerd Ascheid. "A compiler infrastructure for embedded heterogeneous MPSoCs." English. In: *Parallel Computing*. Vol. 40. 2. Elsevier, Feb. 2014, pp. 51–68. doi: `http://dx.doi.org/10.1016/j.parco.2013.11.007`.

[Wei+14] Andreas Weichslgartner, Deepak Gangadharan, Stefan Wildermann, Michael Glaß, and Jürgen Teich. “DAARM: Design-time application analysis and run-time mapping for predictable execution in many-core systems.” In: *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2014, pp. 1–10.

[Cat+15] Vincenzo Catania, Andrea Mineo, Salvatore Monteleone, Maurizio Palesi, and Davide Patti. “Noxim: An open, extensible and cycle-accurate network on chip simulator.” In: *2015 IEEE 26th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE. 2015, pp. 162–163.

[Che+15] Shuang Chen, Xue Lin, Alireza Shafaei, Yanzhi Wang, and Massoud Pedram. “Analysis of deeply scaled multi-gate devices with design centering across multiple voltage regimes.” In: *2015 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. IEEE. 2015, pp. 1–2.

[Kac15] Alexey Kachayev. *Reinventing Haxl: Efficient, Concurrent and Concise Data Access*. Tech. rep. [Online; accessed 4-May-2017]. 2015. url: `https://www.youtube.com/watch?v=T-oekV8Pwv8`.

[Li+15] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. “Gated graph sequence neural networks.” In: *arXiv preprint arXiv:1511.05493* (2015).

[Mel15a] Mellanox Technologies. *TILE-Gx36 Processor*. [Online; accessed 2019-05-22]. Mellanox Technologies. 2015. (Visited on 05/22/2019).

[Mel15b] Mellanox Technologies. *TILE-Gx72 Processor*. [Online; accessed 2019-05-22]. Mellanox Technologies. 2015. (Visited on 05/22/2019).

[Mou+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. “The Lean theorem prover (system description).” In: *International Conference on Automated Deduction*. Springer. 2015, pp. 378–388.

[Pel+15] Maxime Pelcat, Karol Desnos, Luca Maggiani, Yanzhou Liu, Julien Heulot, Jean-François Nezan, and Shuvra S Bhattacharyya. “Models of architecture.” In: (2015).

[QP15] Wei Quan and Andy D Pimentel. “A hybrid task mapping algorithm for heterogeneous MPSoCs.” In: *ACM Transactions on Embedded Computing Systems (TECS)* 14.1 (2015), pp. 1–25.

[Rol+15] Sascha Roloff, David Schafhauser, Frank Hannig, and Jürgen Teich. “Execution-driven parallel simulation of PGAS applications on heterogeneous tiled architectures.” In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2015, pp. 1–6.

[The15] The Multicore Association, Inc. *Software-Hardware Interface for Multi-Many-Core (SHIM) Specification, V1.0.* The Multicore Association, Inc. Jan. 2015.

[Wad15] Philip Wadler. “Propositions as types.” In: *Communications of the ACM* 58.12 (2015), pp. 75–84.

[Bab16] László Babai. “Graph isomorphism in quasipolynomial time.” In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 2016, pp. 684–697.

[Bal+16] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. “OpenPiton: An Open Source Manycore Research Framework.” In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. Atlanta, Georgia, USA: ACM, 2016, pp. 217–232. isbn: 978-1-4503-4091-5. doi: `10.1145/2872362.2872414`. url: `https://doi.org/10.1145/2872362.2872414`.

[Che+16] Kuan-Hsun Chen, Jian-Jia Chen, Florian Kriebel, Semeen Rehman, Muhammad Shafique, and Jörg Henkel. “Task mapping for redundant multithreading in multi-cores with reliability and performance heterogeneity.” In: *IEEE Transactions on Computers* 65.11 (2016), pp. 3441–3455.

[KKM16] Enagnon Cedric Klikpo, Jad Khatib, and Alix Munier-Kordon. “Modeling multi-periodic simulink systems by synchronous dataflow graphs.” In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2016, pp. 1–10.

[Olo16] Andreas Olofsson. “Epiphany-V: A 1024 processor 64-bit risc system-on-chip.” In: *arXiv preprint arXiv:1610.01832* (2016).

[Sha16] Christopher Shaver. “On the Representation of Distributed Behavior.” PhD thesis. EECS Department, University of California, Berkeley, Dec. 2016. url: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-206.html`.

[Sod+16] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu. “Knights Landing: Second-Generation Intel Xeon Phi Product.” In: *IEEE Micro* 36.2 (Mar. 2016), pp. 34–46. issn: 0272-1732. doi: `10.1109/MM.2016.25`. url: `https://doi.org/10.1109/MM.2016.25`.

[Wei+16] Andreas Weichslgartner, Stefan Wildermann, Johannes Götzfried, Felix Freiling, Michael Glaß, and Jürgen Teich. “Design-time/run-time mapping of security-critical applications in heterogeneous mpsocs.” In: *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*. 2016, pp. 153–162.

[Zhu+16] Di Zhu, Lizhong Chen, Siyu Yue, Timothy M Pinkston, and Massoud Pedram. “Providing balanced mapping for multiple applications in many-core chip multiprocessors.” In: *IEEE Transactions on Computers* 65.10 (2016), pp. 3122–3135.

[ABK17] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. “Learning to represent programs with graphs.” In: *arXiv preprint arXiv:1711.00740* (2017).

[AMS17] Josefine Asmus, Christian L Müller, and Ivo F Sbalzarini. “L_p-Adaptation: Simultaneous Design Centering and Robustness Estimation of Electronic and Biological Systems.” In: *Scientific reports* 7.1 (2017), pp. 1–12.

[Cum+17a] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. “Synthesizing benchmarks for predictive modeling.” In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 86–99.

[Cum+17b] Christopher Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. “End-to-end Deep Learning of Optimization Heuristics.” In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2017)*. Portland, Oregon, US, Sept. 2017.

[Kra17] Sebastian Krammer. “Isomorphism-Classes of Subgraphs via Semigroups.” Bachelor’s Thesis. RWTH Aachen, 2017.

[Lee17] Edward A Lee. *Plato and the Nerd: The Creative Partnership of Humans and Technology*. MIT Press, 2017.

[Sch+17] Tobias Schwarzer, Andreas Weichslgartner, Michael Glaß, Stefan Wildermann, Peter Brand, and Jürgen Teich. “Symmetry-eliminating design space exploration for hybrid application mapping on many-core architectures.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.2 (2017), pp. 297–310.

[All+18] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. “A survey of machine learning for big code and naturalness.” In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–37.

[BJH18] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. “Neural code comprehension: a learnable representation of code semantics.” In: *Advances in Neural Information Processing Systems*. 2018, pp. 3585–3597.

[Chi18] David Chisnall. “C is not a low-level language.” In: *Communications of the ACM* 61.7 (2018), pp. 44–48.

[EAC18] Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. “Supporting Fine-grained Dataflow Parallelism in Big Data Systems.” In: *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. PMAM’18. Vienna, Austria: ACM, Feb. 2018, pp. 41–50. isbn: 978-1-4503-5645-9. doi: `10.1145/3178442.3178447`. url: `http://doi.acm.org/10.1145/3178442.3178447`.

[Li+18a] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. “Visualizing the Loss Landscape of Neural Nets.” In: *Neural Information Processing Systems*. 2018.

[Li+18b] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. “Learning deep generative models of graphs.” In: *arXiv preprint arXiv:1803.03324* (2018).

[RG18] Valentina Richthammer and Michael Glaß. “On Search-Space Restriction for Design Space Exploration of Multi-/Many-Core Systems.” In: *MBMV*. 2018.

[Tam+18] S. M. Tam, H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh, S. Vora, and E. Wang. “SkyLake-SP: A 14nm 28-Core xeon® processor.” In: *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*. Feb. 2018, pp. 34–36. doi: `10.1109/ISSCC.2018.8310170`. url: `https://doi.org/10.1109/ISSCC.2018.8310170`.

[Bec19] Micah Beck. “On the Hourglass Model.” In: *Commun. ACM* 62.7 (June 2019), pp. 48–57. issn: 0001-0782. doi: `10.1145/3274770`. url: `https://doi.org/10.1145/3274770`.

[BCJ19] Hasna Bouraoui, Jeronimo Castrillon, and Chadlia Jerad. “Comparing dataflow and openmp programming for speaker recognition applications.” In: *Proceedings of the 10th and 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*. 2019, pp. 1–6.

[Eas+19] James East, Attila Egri-Nagy, James D Mitchell, and Yann Péresse. “Computing finite semigroups.” In: *Journal of Symbolic Computation* 92 (2019), pp. 110–155.

[Ert19] Sebastian Ertel. “Towards Implicit Parallel Programming for Systems.” PhD thesis. Dresden, Germany: TU Dresden, Dec. 2019, 121pp.

[Fet+19] Gerhard Fettweis, Meik Dörpinghaus, Jeronimo Castrillon, Akash Kumar, Christel Baier, Karlheinz Bock, Frank Ellinger, Andreas Fery, Frank H. P. Fitzek, Hermann Härtig, Kambiz Jamshidi, Thomas Kissinger, Wolfgang Lehner, Michael Mertig, Wolfgang E. Nagel, Giang T. Nguyen, Dirk Plettemeier, Michael Schröter, and Thorsten Strufe. “Architecture and Advanced Electronics Pathways Toward Highly Adaptive Energy-Efficient Computing.” In: *Proceedings of the IEEE* 107.1 (Jan. 2019), pp. 204–231. issn: 0018-9219. doi: `10.1109/JPROC.2018.2874895`. url: `https://ieeexplore.ieee.org/document/8565890`.

[Lee19] Edward A Lee. “Freedom From Choice and the Power of Models: in Honor of Alberto Sangiovanni-Vincentelli.” In: *Proceedings of the 2019 International Symposium on Physical Design*. 2019, pp. 126–126.

[LL19] Marten Lohstroh and Edward A Lee. “Deterministic actors.” In: *2019 Forum for Specification and Design Languages (FDL)*. IEEE. 2019, pp. 1–8.

[Tew19] Felix Teweleitt. “A logic language for IoT mappings.” Studienarbeit. TU Dresden, 2019.

[Web19] Matthew Weber. “Context and Interaction in the Internet of Things.” PhD thesis. EECS Department, University of California, Berkeley, Aug. 2019. url: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-114.html`.

[WAL19] Matthew Weber, Ravi Akella, and Edward A Lee. “Service Discovery for the Connected Car with Semantic Accessors.” In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2019, pp. 2417–2422.

[Yad19] Omry Yadan. *Hydra - A framework for elegantly configuring complex applications*. Github. 2019. url: https://github.com/facebookresearch/hydra.

[Zha19] Yong Zhao. "Health monitoring and life-time prognostics to enable dependable many-processor SoCs." PhD thesis. University of Twente, 2019.

[Bra20] Alexander Brauckmann. "Investigating Input Representations and Representation Models of Source Code for Machine Learning." MA thesis. TU Dresden, Feb. 2020.

[BCM20] Nishant Budhdev, Mun Choon Chan, and Tulika Mitra. *Iso-RAN: Isolation and Scaling for 5G RANvia User-Level Data Plane Virtualization*. 2020. arXiv: 2003.01841 [cs.NI].

[CDA20] C/DA - Design Automation. "IEEE Standard for Software-Hardware Interface for Multi-Many-Core." In: *IEEE Std 2804-2019* (Jan. 2020), pp. 1–84. doi: 10.1109/IEEESTD.2020.8985663. url: https://standards.ieee.org/standard/2804-2019.html.

[Cum+20] Chris Cummins, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, Hugh Leather, and Michael O'Boyle. "Program Graphs for Machine Learning." In: *arXiv preprint arXiv:2003.10536* (2020).

[Gat+20] Alan Gatherer, Ashish Shrivastava, Hao Luan, Asheesh Kashyap, Zhenguo Gu, and Miguel Dajer. "Towards a Domain Specific Solution for a New Generation of Wireless Modems." In: *arXiv preprint arXiv:2012.02890* (2020).

[inc20] Kalray inc. *Kalray MPPA3 Coolidge Anouncement*. 2020. url: https://www.kalrayinc.com/release-of-third-generation-mppa-processor-coolidge/.

[KC20] Robert Khasanov and Jeronimo Castrillon. "Energy-efficient Runtime Resource Management for Adaptable Multi-application Mapping." In: *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)*. DATE '20. Grenoble, France: IEEE, Mar. 2020, pp. 909–914. isbn: 978-3-9819263-4-7. doi: 10.23919/DATE48585.2020.9116381. url: https://ieeexplore.ieee.org/document/9116381.

[LC20] Hugh Leather and Chris Cummins. "Machine learning in compilers: Past, present and future." In: *2020 Forum for Specification and Design Languages (FDL)*. IEEE. 2020, pp. 1–8.

[Loh20] Marten Lohstroh. "Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems." PhD thesis. UC Berkeley, 2020.

[Loh+20a] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A Lee. "A Language for Deterministic Coordination Across Multiple Timelines." In: *2020 Forum for Specification and Design Languages (FDL)*. IEEE. 2020, pp. 1–8.

[Loh+20b] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A Lee. “A Language for Deterministic Coordination Across Multiple Timelines.” In: *2020 Forum for Specification and Design Languages (FDL)*. IEEE. 2020, pp. 1–8.

[Nic20] Timo Nicolai. “Faster MPSoC Task Mapping via Symmetry Reduction.” Studienarbeit. TU Dresden, 2020.

[Pal+20] Aditya Paliwal, Sarah M Loos, Markus N Rabe, Kshitij Bansal, and Christian Szegedy. “Graph Representations for Higher-Order Logic and Theorem Proving.” In: *AAAI*. 2020, pp. 2967–2974.

[RFG20] Valentina Richthammer, Fabian Fassnacht, and Michael Glaß. “Search-space Decomposition for System-level Design Space Exploration of Embedded Systems.” In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 25.2 (2020), pp. 1–32.

[Rup20] Karl Rupp. *Microprocessor Trend Data*. https://github.com/karlrupp/microprocessor-trend-data. 2020.

[Thi20] Alexander Thierfelder. “A Domain-Specific Generative Model of Code for LLVM.” MA thesis. TU Dresden, Feb. 2020.

[Ye+20] Guixin Ye, Zhanyong Tang, Huanting Wang, Dingyi Fang, Jianbin Fang, Songfang Huang, and Zheng Wang. “Deep Program Structure Modeling Through Multi-Relational Graph-based Learning.” In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 2020, pp. 111–123.

[BJC21] Hasna Bouraoui, Chadlia Jerad, and Jeronimo Castrillon. “Towards Adaptive multi-Alternative Process Network.” In: *Proceedings of the 12th Workshop and 10th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM’21), co-located with 16th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*. PARMA-DITAM 2021. Budapest, Hungary: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Jan. 2021.

[GAP20] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.11.0*. The GAP Group. 2020. url: %5Curl%7Bhttps://www.gap-system.org%7D.

LIST OF FIGURES

INDEX

LIST OF ACRONYMS

ALU Arithmetic Logic Unit

ANOVA Analysis of Variance

AP Adaptive Platform

API Application Programming Interface

AST abstract syntax tree

BNF Backus-Naur Form

BSGS base and strong generating set

CDFG control- and data-flow graph

cfaed Center for Advancing Electronics Dresden

CAS computer algebra system

CFS Completely Fair Scheduler

CGO International Symposium on Code Generation and Optimization

CPN C for Process Networks

CPU Central Processing Unit

CPS Cyber-Physical System

CSDF Cyclo-Static Data Flow

CSP Communicating Sequential Process

DDF Dynamic Data Flow

DLP data-level parallelism

DMA Direct Memory Access

DPN Dataflow Process Networks

DOL Distributed Operation Layer

DSE Design-Space Exploration

DSL Domain-Specific Language

HAEC Highly-Adaptive Energy-Efficient Computing

E3S Embedded System Synthesis Benchmarks Suite

EDA Electronic Design Automation

FFT Fast Fourier Transform

FIFO first in - first out

FPGA Field Programmable Gate Array

FRP Functional Reactive Programming

GAP Groups Algorithms Programming

GBM Group Based Mapping

GGNN Gated Graph Sequence Neural Network

GPU Graphics Processing Unit

GSM Global System for Mobile Communications

GUI Graphical User Interface

HLS High-Level Synthesis

HOG Histogram of Oriented Gradients

HSDF Homogeneous SDF

IDE Integrated Development Environment

IFFT inverse FFT

i.i.d. independent and identically distributed

ILP integer linear programming

IoT Internet of Things

IP intellectual property

IR intermediate representation

ISA Instruction-Set Architecture

I/O Input/Output

KPN Kahn Process Network

KMQ Kahn-MacQueen

LSTM Long Short-Term Memory

LTE Long Term Evolution

NoC Network on Chip

NVM non-volatile memory

MAPS MPSoC Application Programming Studio

MoC Model of Computation

MPSoC Multi-Processor System-on-Chip

OOP object-oriented programming

OS operating system

PCB printed circuit board

pdf probability density function

PACT International Conference on Parallel Architectures and Compilation Techniques

PE processing element

PHY physical layer

poset partially-ordered set

PRET Precision Timed

pthread POSIX thread

RNTI Radio Network Temporary Identifier

RTM race-track memory

RVC Reconfigurable Video Coding

SADF Scenario-Aware Data Flow

SCC strongly connected component

SDF Synchronous Data Flow

SDF^3 SDF For Free

TETRiS Transitive Efficient Template Run-time System

TGFF task graph for free

UE User Equipment

XML Extensible Markup Language