Efficient Implementation of Resource-Constrained Cyber-Physical Systems Using Multi-Core Parallelism

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Olaf Neugebauer

Dortmund

2018

Ort:	TU Dortmund
Fakultät:	Informatik
Tag der mündlichen Prüfung:	11. Juni 2018
Dekan:	Prof. Dr. Gernot Fink
Gutachter:	Prof. Dr. Peter Marwedel
	Prof. Dr. Heinrich Müller

Acknowledgements

First of all I would like to thank my supervisor Prof. Dr. Peter Marwedel. He provided me the opportunity to pursue the research path leading to this thesis. His initial advice and guidance helped me to orientate and to find my research topic. He offered me to join the ICD to work on the European research project MADNESS where I learned to collaborate with partners from different countries and domains. I would also like to thank Prof. Dr. Heinrich Müller for his commitment as a reviewer for this thesis. He and Prof. Marwedel are responsible for the joint research project between Dr. Pascal Libuschewski and me. Special thanks goes to Prof. Dr. Michael Engel for his advice and support over the years.

Additional thanks to ICD and the European research project for the funding in the beginning of my career. Later parts of this work were performed in cooperation with the Collaborative Research Center 876 founded by the German Research Foundation (DFG) in the context of the subprojects A3 and B2. Further, I like to thank Synopsys for providing the CoMET/Virtualizer simulation platform.

I would also like to thank my colleagues, in particular Dr. Pascal Libuschewski and Dr. Daniel Cordes for the great collaboration. In addition, Roland Kühn and my fellow office mate Helena Kotthaus for their support and fruitful discussions. Finally, I like to thank Hendrik Borghorst, Björn Bönninghoff, Dr. Timon Kelter, Dr. Jan Kleinsorge, Dr. Andreas Heinig, Prof. Dr. Heiko Falk and all other current and former colleagues.

Above all, I would like to thank my friends and family who supported me well beyond this thesis. My mother, who gave me the freedom to join the university and her support during these years. Sebastian and Christoph, friends of my youth, who were always available for non-technical discussions. Thanks to my dog Gizmo, who forced me to take regular breaks and get some fresh air especially during the writing of this thesis. Finally, I would like to thank the most important person in my life, my fiancée and the love of my life Aleksa-Carina Putinas who supported me for almost 17 years in good and bad times. Thank you!

Abstract

The quest for more performance of applications and systems became more challenging in the recent years. Especially in the cyber-physical and mobile domain, the performance requirements increased significantly. Applications, previously found in the high-performance domain, emerge in the area of resource-constrained domain. Modern heterogeneous high-performance MPSoCs provide a solid foundation to satisfy the high demand. Such systems combine general processors with specialized accelerators ranging from GPUs to machine learning chips. On the other side of the performance spectrum, the demand for small energy efficient systems exposed by modern IoT applications increased vastly.

Developing efficient software for such resource-constrained multi-core systems is an error-prone, time-consuming and challenging task. This thesis provides with PA4RES a holistic semiautomatic approach to parallelize and implement applications for such platforms efficiently. Our solution supports the developer to find good trade-offs to tackle the requirements exposed by modern applications and systems. With PICO, we propose a comprehensive approach to express parallelism in sequential applications. PICO detects data dependencies and implements required synchronization automatically. Using a genetic algorithm, PICO optimizes the data synchronization. The evolutionary algorithm considers channel capacity, memory mapping, channel merging and flexibility offered by the channel implementation with respect to execution time, energy consumption and memory footprint. PICO's communication optimization phase was able to generate a speedup almost 2 or an energy improvement of 30% for certain benchmarks.

The PAMONO sensor approach enables a fast detection of biological viruses using optical methods. With a sophisticated virus detection software, a real-time virus detection running on stationary computers was achieved. Within this thesis, we were able to derive a soft real-time capable virus detection running on a high-performance embedded system, commonly found in today's smart phones. This was accomplished with smart DSE algorithm which optimizes for execution time, energy consumption and detection quality. Compared to a baseline implementation, our solution achieved a speedup of 4.1 and 87% energy savings and satisfied the soft real-time requirements. Accepting a degradation of the detection quality, which still is usable in medical context, led to a speedup of 11.1. This work provides the fundamentals for a truly mobile real-time virus detection solution.

The growing demand for processing power can no longer satisfied following well-known approaches like higher frequencies. These so-called performance walls expose a serious challenge for the growing performance demand. Approximate computing is a promising approach to overcome or at least shift the performance walls by accepting a degradation in the output quality to gain improvements in other objectives. Especially for a safe integration of approximation into existing application or during the development of new approximation techniques, a method to assess the impact on the output quality is essential. With QCAPES, we provide a multi-metric assessment framework to analyze the impact of approximation. Furthermore, QCAPES provides useful insights on the impact of approximation on execution time and energy consumption. With ApproxPICO we propose an extension to PICO to consider approximate computing during the parallelization of sequential applications.

Zusammenfassung

Die Leistungsanforderungen von Anwendungen und Systemen sind in den letzten Jahren gestiegen. Gerade im Bereich der Cyber-Physikalischen und mobilen Systemen ist der Bedarf enorm gewachsen. Frühere Hochleistungsanwendungen dringen in den Bereich der Ressourcen-beschränkten Systeme ein. Moderne, leistungsfähige Mehrprozessorsysteme (MPSoC) bieten eine solide Basis, um diesen Leistungsanspruch zu befriedigen. Diese Systeme kombinieren Prozessoren mit speziellen Beschleunigern wie Grafikprozessoren oder auf maschinelles Lernen optimierte Prozessoren. Andererseits steigt der Bedarf nach besonders energieeffizienten System im Bereich des Internet of Things oder der Industrie 4.0 spürbar.

Die Entwicklung effizienter Software für diese Ressourcen-beschränkten Mehrkernsysteme ist eine fehleranfällige, zeitaufwendige und komplexe Aufgabe. Diese Arbeit bietet mit PA4RES einen einheitlichen, semiautomatischen Ansatz, um Anwendungen für solche Plattformen effizient zu parallelisieren und zu implementieren. Unsere Lösung unterstützt Entwickler in dem Abwägungsprozess zwischen den verschiedenen Zielkriterien der Anwendung im Hinblick auf eine Zielplattform. Mit PICO stellen wir einen umfassenden Ansatz vor, der es ermöglicht Parallelität in sequentiellen Anwendungen einfach auszudrücken. Dabei erkennt PICO automatisch Datenabhängigkeiten und fügt notwendige Synchronisierungsoperationen automatisch ein. Die durch PICOs Kommunikationsoptimierung generierten Lösungen erzielen in einigen Fällen nahezu eine Beschleunigung um den Faktor 2 und eine Energieeinsparung um 30%. Der evolutionäre Optimierungsalgorithmus betrachtet die Kapazität, das Speicherlayout, die Zusammenlegung und die Flexibilität in der Implementierung von Kommunikationskanälen unter Berücksichtigung der Laufzeit, des Speicherbedarfs und des Energieverbrauchs.

Der PAMONO Sensor ermöglicht eine schnelle Erkennung von biologischen Viren mit Hilfe eines optischen Verfahrens. Eine hochentwickelte Detektionssoftware ermöglicht eine echtzeitfähige Viruserkennung auf stationären Computern. In dieser Arbeit konnten wir eine echtzeitfähige Version der Software generieren, die auf typischen Smartphone-Prozessoren läuft. Dies wurde unter Zuhilfenahme intelligenter Entwurfsraumexplorationsmethoden unter Berücksichtigung von Laufzeit, Energieverbrauch und Erkennungsgüte erreicht. Im Vergleich zu der Basisversion erreichte unsere Lösung eine Beschleunigung

um den Faktor 4.1 und eine Energieeinsparung um 87%. Wenn man eine Verringerung der Erkennungsrate erlaubt, die immer noch ein verwendbares Ergebnis im medizinischen Kontext bedeutet, erzielt unser Verfahren eine Beschleunigung um den Faktor 11.1. Diese Arbeit legt damit den Grundstein für einen zukünftigen batteriebetriebenen Einsatz des Viruserkennungssystems.

Der steigende Bedarf nach noch mehr Leistung kann absehbar nicht mehr mit klassischen Ansätzen wie der Erhöhung der Taktfrequenz befriedigt werden. Diese Beschränkungen bedeuten ein ernsthaftes Problem, um den Bedarf an Leistung zu erfüllen. *Approximate Computing* ist ein vielversprechender Ansatz um diese Beschränkung zu umgehen oder zumindest aufzuweichen. Hier wird bewusst eine Verringerung der Ausgabequalität in Kauf genommen, um eine Leistungssteigerung zu erzielen. Gerade die Bestimmung der Auswirkungen durch *Approximate Computing* verlangt einen zuverlässigen, automatisierten Ansatz. Mit QCAPES haben wir ein mehrkriterielles Analyseframework geschaffen, um die Auswirkungen von *Approximate Computing* ganzheitlich zu betrachten. Weiterhin bietet QCAPES einen zusätzlichen Einblick in die Auswirkungen durch die approximierte Anwendung auf die Laufzeit und den Energieverbrauch. Mit ApproxPICO stellen wir unseren Ansatz vor *Approximate Computing* während der Parallelisierung von sequentiellen Anwendungen zu betrachten.

Publications

Olaf Neugebauer, Michael Engel, and Peter Marwedel. “A Parallelization Approach for Resource-Restricted Embedded Heterogeneous MPSoCs Inspired by OpenMP”. In: *Journal of Systems and Software (JSS)* 125 (2016), pp. 439–448. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/j.jss.2016.08.069

Olaf Neugebauer, Michael Engel, and Peter Marwedel. “Multi-Objective Aware Communication Optimization for Resource-Restricted Embedded Systems”. In: *Proceedings of Architecture of Computing Systems (ARCS)*. 2015

Olaf Neugebauer, Pascal Libuschewski, Michael Engel, Heinrich Müller, and Peter Marwedel. “Plasmon-based Virus Detection on Heterogeneous Embedded Systems”. In: *Proceedings of Workshop on Software & Compilers for Embedded Systems (SCOPES)*. 2015, pp. 48–57. ISBN: 978-1-4503-3593-5. DOI: 10.1145/2764967.2764976

Olaf Neugebauer, Michael Engel, and Peter Marwedel. “A Parallelization Approach for Resource-Restricted Embedded Heterogeneous MPSoCs Inspired by OpenMP”. In: *The 1st ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems (SEPS)*. 10/2014

Olaf Neugebauer, Peter Marwedel, Roland Kühn, and Michael Engel. “Quality Evaluation Strategies for Approximate Computing in Embedded Systems”. In: *Technological Innovation for Smart Systems*. Ed. by Luis M. Camarinha-Matos, Mafalda Parreira-Rocha, and Javaneh Ramezani. Vol. 499. Cham: Springer International Publishing, 2017, pp. 203–210. ISBN: 978-3-319-56077-9

Olaf Neugebauer, Michael Engel, and Peter Marwedel. “Approximate Communication in Parallel Applications for Resource-Constrained Embedded Systems”. In: *Workshop on Approximate Computing*. 10/2015

Daniel Cordes, Michael Engel, Peter Marwedel, and Olaf Neugebauer. “Automatic extraction of multi-objective aware pipeline parallelism using genetic algorithms”. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*. 10/2012

Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. "Automatic Extraction of Multi-Objective Aware Parallelism for Heterogeneous MPSoCs". In: *Proceedings of the Sixth International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*. 09/2013

Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. "Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs". In: *Proceedings of the Fourth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*. 10/2013

Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. "Automatic Extraction of Pipeline Parallelism for Embedded Heterogeneous Multi-Core Platforms". In: *Proceedings of the Sixteenth International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*. 10/2013

Helena Kotthaus, Andreas Lang, Olaf Neugebauer, and Peter Marwedel. "R goes Mobile: Efficient Scheduling for Parallel R Programs on Heterogeneous Embedded Systems". In: *Abstract Booklet of the International R User Conference (UseR!)*. 07/2017

Peter Marwedel, Heiko Falk, and Olaf Neugebauer. "Memory-Aware Optimization of Embedded Software for Multiple Objectives". In: *Handbook of Hardware/Software Codesign*. Ed. by Soonhoi Ha and Jürgen Teich. Springer Netherlands, 2017. ISBN: 978-94-017-7358-4. DOI: 10.1007/978-94-017-7358-4_27-2

Emanuele Cannella, Lorenzo Di Gregorio, Leandro Fiorin, Menno Lindwer, Paolo Meloni, Olaf Neugebauer, and Andy D. Pimentel. "Towards an ESL Design Framework for Adaptive and Fault-tolerant MPSoCs: MADNESS or not?" In: *Proceedings of the 9th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. 10/2011

In addition, the author contributed to the yearly graduate school reports of the Collaborative Research Center 876: [MR13; MR14; MR15; MR16; MR17]

Contents

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CPS	Cyber-Physical System
CPU	Central Processing Unit
CRC	Collaborative Research Center
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
DFS	Dynamic Frequency Scaling
DSE	Design Space Exploration
DSP	Digital Signal Processor
ECJ	Java-based Evolutionary Computation Research System
FEHLER	Flexible Error Handling in Embedded Real-Time Systems
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
HPC	High-Performance Computing

ILP	Integer Linear Programming
IoT	Internet of Things
IR	Intermediate Representation
ISA	Instruction Set Architecture
KPN	Kahn Process Network
MADNESS	Methods for predictAble DesigN of heterogeneous Embedded Systems with adaptivity and reliability Support
MAE	Mean-Absolute Error
MNEMEE	Memory management technology for adaptive and efficient design of embedded systems
MPSoC	Multiprocessor System-on-a-Chip
MSE	Mean-Squared Error
MSSIM	Mean Structural Similarity Index Metric
NoC	Network-on-Chip
PA4RES	Parallelization for (4) Resource-restricted Embedded Systems
PAMONO	Plasmon-Assisted Microscopy of Nano-Objects
PAXES	Parallelism Extraction for Embedded Systems
PDG	Program Dependence Graph
PICO	Parallelism Implementer and Communication Optimizer
PSNR	Peak-Signal-to-Noise Ratio
PQVM	Perceptual Visual Quality Metrics
QCAPES	Quality Comparison for Approximate Programs on Embedded Systems
QoR	Quality of Results
QoS	Quality of Service
RAW	Read-After-Write
RMSE	Root-Mean-Squared Error
RTEMS	Real-Time Executive for Multiprocessor Systems
SFM	Signal Fidelity Metrics

SLOC Source Lines Of Code

SoC System-on-a-Chip

SPM Scratch Pad Memory

SSIM Structural Similarity Index Metric

UIQI Universal Image Quality Index

WAR Write-After-Read

WAW Write-After-Write

Chapter 1

Challenges in Embedded Cyber-Physical Systems

1.1 Introduction

Computer systems are an integral part of today's life. They do various tasks, starting from basic functions like counting objects passing a light barrier to very complex tasks like an automatically generated personalized drug therapy. It is almost impossible to find a domain completely free of computers. Estimations predict that we will see between 30 billion [IDC15] and 50 billion [Eva11] connected Internet of Things (IoT) devices in 2020. Beside creating new applications from scratch, developers will face the task to adapt legacy code to these systems. Often, existing sequential software should be reused or existing algorithms should be adapted to reduce the time to market. Energy consumption and the trend to green computing are important aspects of today's computer design. To tackle these challenges, new approaches are necessary which take the requirements and restrictions like limited energy consumption into account.

Embedded systems are used in a variety of domains like automotive, avionics, manufacturing and health care. Marwedel [Mar17] defines embedded systems as "information processing systems embedded into enclosing products". Those systems are covering a wide range of the available technology. For instance, the compute elements of these systems range from very simple processors with extremely small memories to heterogeneous multiprocessor systems. Whereas some systems like those built into airbags are highly specialized and use special components like Digital Signal Processors (DSPs), others, for instance smart phones, use more generalized processors. The potential capabilities of embedded systems are often dictated and limited by the environments they are embedded in. For example, a mobile usage relies on battery power and thus the system needs to be very energy efficient. The term Cyber-Physical Systems (CPSs) is used to highlight the importance of the physical interaction in those systems. According to Lee [Lee07], "Cyber-Physical Systems are integrations of computation and physical processes". Despite the limitations, today's application's demands are increasing. For instance, the trend

is to combine CPSs with a certain kind of intelligent algorithms like machine learning, exposing new challenges to the designers. Apple's Core ML machine learning framework [App17] enables the usage of sophisticated machine learning models across their entire product range, leading to billions of machine learning capable mobile phones. In general, CPSs build the foundation of smart systems and the IoT. Thus, "applications of CPS arguably have the potential to dwarf the 20-th century IT revolution" and "the most interesting and revolutionary cyber-physical systems will be networked" [Lee08].

In smart systems, potentially dozens of different sensors are combined and their data are fused. With this new knowledge, the system analyzes its state and tries to calculate the next steps e.g. performed by actuators. This is known as hardware in the loop [Mar17]. Predicting the future can be challenging, thus smart algorithms like machine learning are emerging into this domain exposing new demands to CPS. In the automotive or aviation sector, a collision avoidance system needs multiple sensors, providing data like current speed, direction of travel and radar ranging information. A central compute instance uses the information to predict the likelihood of a collision with an object. In case of a potential collision the system reacts to avoid the collision. Possible reactions vary from simple audible warnings to changing the direction of travel.

The next step in the digital evolution is truly the connection of multiple systems. Here, not only classic computer systems are used, the vast majority will be CPSs, leading to an IoT. Kopetz [Kop11] describes this as systems of systems. As mentioned before, predictions say that we will see between 30 billion [IDC15] and 50 billion [Eva11] connected IoT devices in 2020. McKinsey [MCB15] predicts a potential impact of $3.9 trillion to $11.1 trillion per year in 2025 from the IoT sector. One example is Industry 4.0 [KRB14], summarizing the trends in factories to combine automation, data exchange and analysis. According to this paradigm, the modern manufacturing process will make intensive use of connected CPSs and smart systems building an IoT. An exemplary use case might be a welding robot reporting its health status to a central computer system. Maintenance teams can use this data to optimize the maintenance schedule. If each component of the production process is able to provide data, big data analytics can be used to extract new knowledge to further optimize the production process or to predict future failures and thus increase quality. The Airbus Skywise data platform provides a predictive maintenance system for Airbus airplanes [Air18]. Skywise is a commercial service which monitors components health and in conjunction with big data analysis enables engineers to intervene early and replace parts before a failure.

To enable these new trends, this thesis focuses on the fundamental question: how can we satisfy the increasing demand of today's applications with current resource-constrained Cyber-Physical Systems? In particular, we focus on the software development process for these systems. The definitions of CPS and IoT are very similar and often mixed, especially when talking about the devices (things). Thus, in this thesis we will use CPS, IoT and embedded systems similarly but emphasize differences if necessary.

In the following Section 1.2 we discuss some common challenges and requirements

exposed by CPS and applications. A biological virus detector is used as a motivating example in Section 1.3. Section 1.4 briefly summarizes the contributions of this dissertation, followed by an outline of this thesis is given in Section 1.5. Finally, Section 1.6 lists all contributing authors to this thesis.

1.2 Challenges of Embedded Software Design

Designers of Cyber-Physical Systems are typically facing a large variety of challenges they need to solve to meet their requirements. According to [Mar17], CPSs need to be among others dependable, timing and resource-aware. For example, the availability of a service might be of high importance and to ensure a certain availability in case of high workload, the systems might reduce the output quality or consume more energy. Furthermore, users of CPSs expect the system not to crash since those systems are usually expected to start once and run forever. In addition, Rajkumar et al. [RLS10] propose several challenges like the composition of several CPSs, the computational and timing abstraction.

The list of properties and challenges of CPS design is long. Thus, in this thesis we focus on the following key challenges:

Energy consumption: A huge amount of embedded systems are powered by batteries or other limited sources of electrical power like solar cells. Even if the systems have a constant power source, limitations exposed by their embedded nature can limit the allowed power consumption or the related thermal dissipation. As some sensors require a cool atmosphere to function properly, integrating a powerful and potentially hot embedded system will prevent the sensor from working properly. Finally, power generation causes costs and reducing the overall energy consumption will reduce the financial expenses.

Execution time: Embedded systems are often subject to timing requirements and need to finish execution before a certain deadline. For example, a system has to produce a valid output every second. In general one can distinguish between hard and soft deadlines. Missing a hard deadline will lead to fatal consequences. Airbags in a car expose a natural hard deadline. Missing the deadline to inflate the airbag might lead to loss of life. All other timing constraints are soft deadlines.

Memory consumption: Typical CPSs only provide a very limited amount of memory. In addition, some embedded systems use special memories that are very small and energy efficient. Especially during application development, those properties need to be considered and exploited.

Output quality or Quality of Service: Modern applications of embedded systems like pattern recognition or multimedia processing allow the relaxation of the output quality and thus reducing their Quality of Service to improve in other objectives. For instance, adaptive video compression codecs trade quality with

memory consumption. Other mobile video codecs reduce the output quality in order to reduce the energy consumption. In general, output quality can be traded against energy consumption, execution time and memory consumption. Nevertheless, applications might specify a minimum quality level.

System designers traditionally tackle these challenges from the hardware and software side. However, in this thesis we focus on approaches aiding the software development side of CPSs assuming a fixed hardware platform. Here, the overall structure of the platform like number of processors, available memory, battery capacity is fixed but can be configured to a certain degree like the frequency and supply voltage. This assumption is based on the observation that more and more standardized hardware platforms are used. For example, ARM-based platforms dominate the mobile phone market and thus are comparatively cheap and often considered in other areas like medical applications as well. According to ARM [ARM17], ARM owns a market share of 90% in mobile application processors in 2016. For 2025, ARM sees itself as the leading company enabling the success of IoT. This thesis will make extensive use of ARM-based systems. However, most presented approaches are platform agnostic and thus not limited to ARM-based systems.

Developers are burdened to find the perfect balance between these challenges to meet the requirements effectively in a cost efficient way. Unfortunately, in most cases, there is no *best* solution satisfying all requirements but rather multiple solutions. Some of them are called Pareto-optimal:

Definition 1.1:
(informal) A solution is called ***Pareto-optimal*** *if there is no other solution which is inferior in no objective and is better in at least one objective.*

Manual approaches to find these solutions are complex, erroneous and time consuming, thus new holistic methodologies considering all aspects are required. This thesis provides methodologies, tools and directions to aid software developers achieving the requirements.

1.3 Motivating Example: Plasmon-based Virus Detection

Cyber-Physical Systems are used in several domains like automotive, aviation or fabrication, see [Mar17] for more examples. In this section, we present an exemplary application and future vision of CPSs. Detection of biological viruses is classically done in stationary facilities like hospitals. In remote locations, samples are usually collected and shipped to a central laboratory. This leads to a potentially long timespan between incubation, first symptoms and proper medical treatment. To reduce this gap, new detection techniques and portable solutions are required. Within the Collaborative Research Center (CRC) 876 “Providing Information by Resource-Constrained Data Analysis”, project B2 [SFB17a], we developed a virus detection solution achieving a soft real-time capable

mobile battery powered virus detection. Further, we also provide the fundamentals for a future distributed sensor network usage.

The basic idea of the sensor system is to capture an interaction process of a virus with a specific environment using a video camera. A software detection pipeline processes the captured images and returns the number of detected viruses. This process is called Plasmon-Assisted Microscopy of Nano-Objects (PAMONO). To enable a mobile usage on a specific hardware platform, the algorithms need to be adapted to meet certain energy consumption limits. In addition, to ensure a fast virus detection, certain execution time requirements must be met. Storing images can be beneficial to improve the detection quality, but due to limited memory capacity this is not always possible. Some parameters of the detection algorithm can trade execution time with a degradation of the output quality. This might result in an increased number of wrong virus detections. In cases where it is not necessary to detect all viruses or where it is acceptable to have some non-viruses classified as viruses (false positive) this property can be used to tune the application parameters to fulfill the performance demands. Finding a good configuration in thousands of different software and hardware configurations is a complex task. Thus, we used smart algorithms to explore the solution space and find proper configurations. We exceeded the original expectations by satisfying all requirements without limitations. In Chapter 5, we give a detailed description of the sensor application and the conducted optimizations.

The next step towards an aerial virus detection system with dozens of sensors is to move at least the preprocessing to a low-power embedded system closer to the sensor. For example, such a system can be deployed at airports to detect and control the outbreak of an epidemic. This application scenario raises new interesting questions. Most obviously, the whole sensor system needs to be miniaturized. Further, offloading of computation might be necessary. In this scenario, the data acquisition is spatially separated from the actual detection pipeline. Inexpensive low-power embedded systems could be used for the data acquisition, transmitting data to a high-performance computer system running the detection application. However, the questions are: How can the existing processing pipeline be modified in order to be executed on these systems? Which part of the pipeline can be offloaded? Which part can be efficiently executed on low-power embedded systems? What data needs to be transmitted?

This example application combines classic embedded system properties like resource-restriction mixed with new high-demand algorithms like pattern recognition, typical for modern CPS and IoT applications. Similar systems can be found in autonomous cars, health care, industry and aviation. In this thesis we conducted our experiments and evaluations with other applications and algorithms as well but we will often refer to the presented virus detection example as the overall picture.

1.4 Contribution of this Work

This thesis provides methodologies, tools and directions for software developers facing the newly exposed requirements of CPSs. Starting with a legacy sequential application, we provide a semi-automatic parallelizing methodology taking the resource restrictions and properties of modern heterogeneous embedded platforms into account. This approach primarily targets low-power embedded systems. Those systems are typically very restricted in their capabilities and for example have no or limited operating system support. This work is internally summarized in the Parallelization for (4) Resource-restricted Embedded Systems (PA4RES) project. Two European Union FP7 research projects contributed to PA4RES. The Memory management technology for adaptive and efficient design of embedded systems (MNEMEE) [MMB11] project builds the fundamentals for the parallelization methodology, the model of computation and the simulator-based platforms. In contrast, technical realizations are partly developed in the Methods for predictAble DesigN of heterogeneous Embedded Systems with adaptivity and reliability Support (MADNESS) project [CGF11].

The author of this thesis provides an integral part of the PA4RES framework with Parallelism Implementer and Communication Optimizer (PICO). It enables an efficient expression of parallelism in sequential applications. The key aspect is that PICO keeps the original code which increases maintainability of the application. Complex types of parallelism can be expressed using abstract high-level methods. In addition, PICO supports the extraction of pipeline parallelism in loops. This enables the parallelization of loops which previously could not be parallelized due to data dependencies. Using hybrid pipeline parallelism, the performance of the parallel application can further be improved. PICO's static data analysis extracts data dependencies between parallel regions and adds communication automatically. The communication model keeps the sequential semantics of the original application. Using evolutionary algorithms, PICO optimizes these data exchanges with respect to available memory, energy consumption and execution time. We propose a high-level abstract execution model which keeps the essential parts of the parallel application which reduced the evaluation time drastically by still obtaining usable results.

By using a high-performance embedded system and smart software optimization techniques, we are able to provide fundamental work enabling a mobile and distributed biological virus detection. This virus detection was previously only possible on powerful computers and it was doubtful if it can be executed on a mobile platform. This work was done in the CRC 876 "Providing Information by Resource-Constrained Data Analysis", project B2 [SFB17a]. The optimization algorithm is able to explore software and a real hardware platform efficiently considering the objectives energy consumption, execution time and detection quality. We are able to meet the soft real-time requirements without losing accuracy. However, solutions with a reduced detection quality, which are still usable in a medical context, achieved a speedup of more than 11 and an energy saving of 94% compared to the baseline. This led us to consider approximate computing as a new

technique to further increase application's performance.

To analyze the impact of approximate computing on the performance of applications especially considering resource-constraints exposed by CPSs, we developed the Quality Comparison for Approximate Programs on Embedded Systems (QCAPES) assessment framework. Embedded in PA4RES, QCAPES enables a fast and automatic assessment of approximation techniques regarding execution time, energy consumption, output quality and other objectives like file size. One key feature is that QCAPES provides a sophisticated multi-metric and multi-objective assessment. Using multiple metrics can help to qualify the accuracy loss in detail. In addition, it can give indications of the error type in the output. With quality case studies we revealed new insights into approximate computing especially in the CPSs domain. We observed unwanted side effects like increased file size when using prominent approximation techniques. In cases of distributed embedded systems, this observation can render the benefits from approximate computing useless since larger files needs to be transmitted at increased cost. The QCAPES assessment framework is available under an open source license and provides a clear interface for extensions.

Adding approximation computing to existing application is a complex and error-prone task. Done improperly, the benefits of approximation might be overshadowed by unwanted side effects. This ranges from unacceptable results to application crashes. Therefore, we provide with ApproxPICO the fundamentals for a holistic approach to integrate approximation techniques into the parallelization process. Furthermore, this thesis developed two open source energy measurement applications targeting the Odroid-XU3 platform.

1.5 Outline

This thesis is structured as follows:

Chapter 2 introduces the PA4RES methodology, our approach to parallelize sequential applications for resource-restricted embedded systems. Furthermore, Chapter 2 provides an overview of supported types of parallelism and the target platforms in combination with the abstract system model. In addition, this chapter presents the parallelizer PAXES and Performance Estimator integrated in PA4RES.

Chapter 3 presents our method to identify parallel regions in sequential applications. Thereby this chapter focuses on the parallelism implementation side of PICO. Using an application model, PICO automatically generates a parallel version of a formerly sequential application and takes care of necessary data synchronization.

Chapter 4 introduces the PICO communication optimization approach. Using an evolutionary algorithm, PICO explores various implementation and mapping parameters

to optimize the inevitable data synchronization. This chapter proves the effectiveness of this approach with several benchmarks in a simulation- and hybrid model-based evaluation.

Chapter 5 presents the PAMONO-based virus detection pipeline and our work to execute this demanding application on a high-performance embedded system. With an evolutionary algorithm, we explored several software and hardware-related parameters in order to obtain an optimized soft real-time capable solution. With our work, we provided the foundation for a fast and mobile virus detection solution.

Chapter 6 focuses on approximate computing, a method to improve applications' performance in trade-off with a loss in output quality. With qualitative case studies, we emphasize the importance of our multi-objective framework QCAPES. Finally, this chapter presents ApproxPICO, an extended version of PICO to consider approximation during the parallelization process.

Chapter 7 concludes this thesis and provides directions for future work.

Appendix A provides additional information regarding the energy model used by the simulator and energy measurement software. Furthermore, the appendix gives a detailed description of the PICO Application Programming Interface (API). Finally, this part provides measurement results used during the communication model tuning.

Appendix B lists additional results for the experiments conducted in Chapter 4.

1.6 Author's Contribution to this Dissertation

According to §10(2) of the "Promotionsordung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011", a dissertation within the context of doctoral studies has to contain a separate list that highlights the author's contributions to research and results obtained in cooperation with other researchers. In the following, we list all chapters and contribute all authors:

Chapter 2: The PA4RES framework bases on several tools. The parallelism extraction tool PAXES was developed by Daniel Cordes [Cor13]. The author of this thesis provided technical support and with PICO, today's back-end of the PAXES parallelization flow. Together we published the work in [CEN13c; CEN13b; CEN13a; CEM12]. The PA4RES framework uses the simulation-based platforms including software support developed by Andreas Heinig [Hei10]. All major parts of PA4RES built upon the MACCv2 framework developed by Robert Pyka [Pyk17]. Florian Schmoll and the author of this thesis created the energy model used by the simulator.

Chapter 3: PICO uses the ICD-C compiler infrastructure, developed, maintained and extended by many doctoral students of our department. Together with Florian Schmoll, the author of this thesis integrated a sophisticated points-to and alias analysis into the compiler framework. PICO was solely developed by the author of this thesis. A first draft of the communication API was published in [CGF11]. However, Stefan Noll [Nol14] and Roland Kühn [Küh16] provided with their Bachelor's theses, supervised by the author of this thesis, additional test cases. Results of this chapter were published in [NEM14; NEM16].

Chapter 4: The communication optimization was solely developed by the author of this thesis. Later, Roland Kühn, as a student employee, helped with evaluation and tuning of the execution model. Work of this chapter was published in [NEM15a].

Chapter 5: Prof. Marwedel suggested to consider the PAMONO virus detection pipeline as a benchmark. The resulting collaboration between Pascal Libuschewski and the author of this thesis is presented in this chapter. Both authors contributed equally to the published paper [NLE15]. The author of this thesis provided the initial idea to port the virus detection software to the Odroid-XU3. In addition, the author provided the system and developed parts of the software support for the measurement on the real hardware. Pascal Libuschewski provided the interface to the optimizing algorithm, bug fixes and conducted the experiments. Selected results from this chapter were also published in textbooks [MFN17; Mar17].

Chapter 6: The technical implementation of QCAPES and the practical evaluation was conducted by Roland Kühn in his Bachelor's thesis. The author of this thesis is responsible for the general idea and guided the development significantly. The results were published in [NMK17]. The ApproxPICO extension is proposed and implemented by the author of this thesis.

Appendix A: Florian Schmoll and the other of this thesis created the energy model used by the simulator. The EnergyMeter was derived from an existing solution by the author of the thesis, whereas the Energy Relay Reader was developed by Alexander Lochmann, Roland Kühn and the author of this thesis.

Besides the before mentioned publications, the author of this thesis published parts of this thesis in the yearly graduate school reports of the Collaborative Research Center 876: [MR13; MR14; MR15; MR16; MR17]. The author of this thesis provided technical support and the hardware for the joint publication with Helena Kotthaus [KLN17].

Chapter 2

Utilizing modern MPSoCs - The PA4RES Methodology

Contents

The quest for more performance was traditionally solved by increasing the frequency of the processors. Due to energy and thermal restrictions, this strategy reached its limits and hits the power wall. This is obvious since the breakdown of Dennard's scaling [DGR74] around 2006. Thus, to further improve the performance, systems designers added more, potentially simpler, processors and thus try to increase the overall performance by exploiting parallelism. If a system is built onto a single chip it is called a Multiprocessor System-on-a-Chip (MPSoC). An existing, large code base of legacy sequential software must be adapted to benefit from these new systems. This chapter provides a general overview of the considered system architectures, especially the systems used in this thesis. In addition, this chapter presents fundamental principles of parallelism used to explore the capabilities of modern systems, including a discussion of the challenges encountered during parallelization of sequential code. Finally, the chapter presents

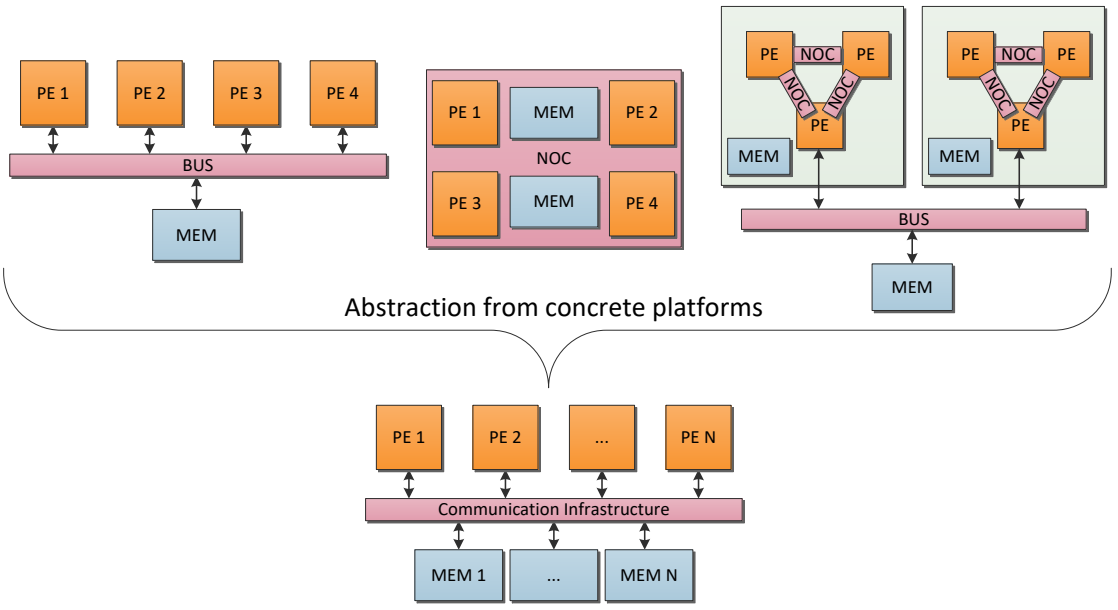


Figure 2.1: Abstract system model of modern embedded systems.

the PA4RES methodology. Our approach tackles the quest for more performance in a resource constrained multiprocessor environment focusing on parallelization of sequential (legacy) code.

2.1 System Architecture Overview

Today's embedded systems combine several processing elements (PE) like general purpose processors, Graphics Processing Units (GPUs) or DSPs. If all elements are equal these systems are called *homogeneous*. Systems with different elements or the same elements with different configurations are called *heterogeneous*. Beside the processing elements, a variety of different memory types like fast and small Scratch Pad Memories (SPMs) or large DRAMs are used. In addition, some systems provide hardware support for direct data exchange like hardware First In First Out (FIFO) channels. Others may use a common bus to connect the elements of the system.

Definition 2.1 (PA4RES System Model):
A system S is composed of a set of processing elements $pe \in PE$ and a set of memories $m \in MEM$ connected through a set of communication interconnects $ci \in CI$. A memory or interconnect may be exclusively assigned to a processing element.

For instance, the Texas Instruments Keystone architecture [Tex18] provides multiple inter- and intra-device communication facilities. In addition, this architecture may provide multiple memory hierarchy levels which can be exploited for communication purposes. Figure 2.1 gives a graphical representation of the abstract system model used in this thesis. Internally, PA4RES abstracts from concrete systems which use buses, Network-on-Chips (NoCs) or hybrid approaches. However, in the model, a communication

infrastructure connects processing elements with possible point-to-point connections and sets of memories. Developing software efficiently for those systems can be challenging. Especially in the context of CPS, a lot of sequential legacy code exists and should be reused. In this thesis, we evaluate our methods on exemplary simulator-based and real hardware systems following this abstract system model.

2.1.1 Simulator-based Low-Power Systems

Simulation-based evaluation offers several benefits compared to real hardware platforms. Simulators enable the simulation of (not yet existing) prototype hardware platforms or configurations. Simulators are deterministic, thus every execution leads to the same behavior which simplifies the evaluation. In real hardware, for instance, variation in the manufacturing process can lead to slightly different timing behaviors. Simulators can be cheaper than real hardware, especially in the case of specialized systems with a small production volume. Traditionally, simulators are orders of magnitudes slower than real hardware. Parallel execution of multiple simulator instances can speed up the overall evaluation time. However, several existing approaches increase the simulation speed of a single instance. For example, using higher abstraction level models will reduce the simulation time by reducing the precision of the simulation. Hybrid approaches combine precise and abstract models, for example, a cycle-accurate processor simulator with a basic memory simulator which just returns the memory values without simulating the internals of the memory. In addition, the simulation results highly depend on the correctness of the underlying model. If the model is faulty or imprecise, it will directly influence the quality of the simulation results.

The simulation-based systems used in this thesis are simulated with Synopsys' Virtualizer [Syn17]. The system comprises four ARM 1176 processors and several memories connected through a bus. These platforms have been used within the MNEMEE [MMB11] and Flexible Error Handling in Embedded Real-Time Systems (FEHLER) [FEH18] projects and several theses [Cor13; Hei15; Kel15; Pyk17; Hol17]. Andreas Heinig did most of the adaptation and provision of the platforms. The author of this thesis refined the energy model used by these systems. Energy values are obtained from CACTI [MBJ09] for memories. CACTI provides an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. Processor values are derived from a self-developed high-level frequency-dependent energy model based on an existing model [SKW01]. Appendix A.1 provides detailed information of the energy model and the integration into the simulation environment. This simplified model is applicable since in this thesis all optimization techniques are only compared on the same hardware platform. Thus, inaccuracies should be reflected in the original and optimized version equally.

On each processor of the simulation-based system, a Real-Time Executive for Multiprocessor Systems (RTEMS) [RTE16] operating system instance is running. Thus, it is more like a distributed system and not a true multiprocessor operating system managing

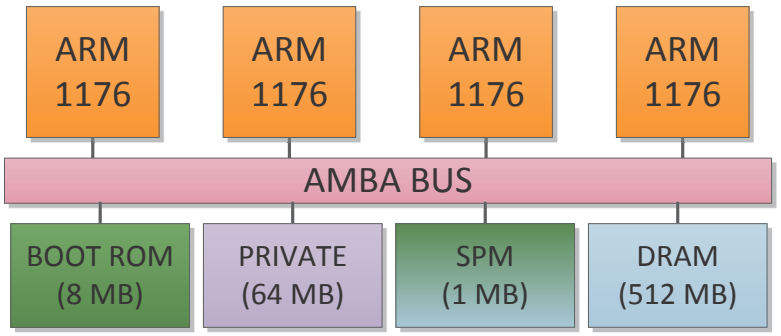


Figure 2.2: Simulator-based quad core embedded MPSoC.

multiple cores in a single instance. This scenario reflects that sometimes processors from different manufacturers, with their own operating system, are combined into a single system. Mixed criticality systems follow a similar approach to isolate critical and non-critical components. R^2G [Hei10] is used as a lightweight library-based middleware, simplifying, among others, task creation and memory allocation for the target system with its multiple operating systems. As common in low-power embedded systems, task mapping and memory allocation is done statically.

All platform versions use the same memory architecture with four different memories. A 8 MB boot ROM is used for the boot loader exclusively, a 1 MB SPM is partially used by the operating systems. The remaining space of the scratch memory can be used by our runtime library or the developer. A partitioned 64 MB DRAM is used to store processor-dependent data privately. The SPM is connected to the bus and can be utilized by each processor and thus be used e.g. for communication between the cores. Finally, a 512 MB DRAM is used for large data objects. The memories and the processors are connected through a bus. Figure 2.2 shows the overall platform structure. In the homogeneous case, all cores are running at a frequency of 500 MHz. In the heterogeneous case the clock frequency varies between 100 and 500 MHz. Later chapters of this thesis, especially for the experiments, will present details on the specific platform configurations.

2.1.2 Real Hardware High-Performance System

As an exemplary real hardware high-performance system we selected the Odroid-XU3 development board from Hardkernel [Har16]. This board uses an Exynos-5422 MPSoC which is also used in millions of Samsung S5 smart phones. The Odroid-XU3 uses the ARM big.LITTLE [Pet13] architecture where processors running at different clock frequencies, featuring differing pipeline structures, cache sizes, bus systems, or memory hierarchies are combined into a single chip. By utilizing the different capabilities efficiently, energy and performance trade-offs can be found to satisfy the requirements of mobile embedded devices and their applications.

Figure 2.3 gives an overview of the Odroid-XU3 platform. An Exynos-5422 octa-core processor with four ARM Cortex-A15 and four Cortex-A7 processors is the main component of the Odroid-XU3. In addition, a Mali-T628 MP6 GPU with full OpenCL

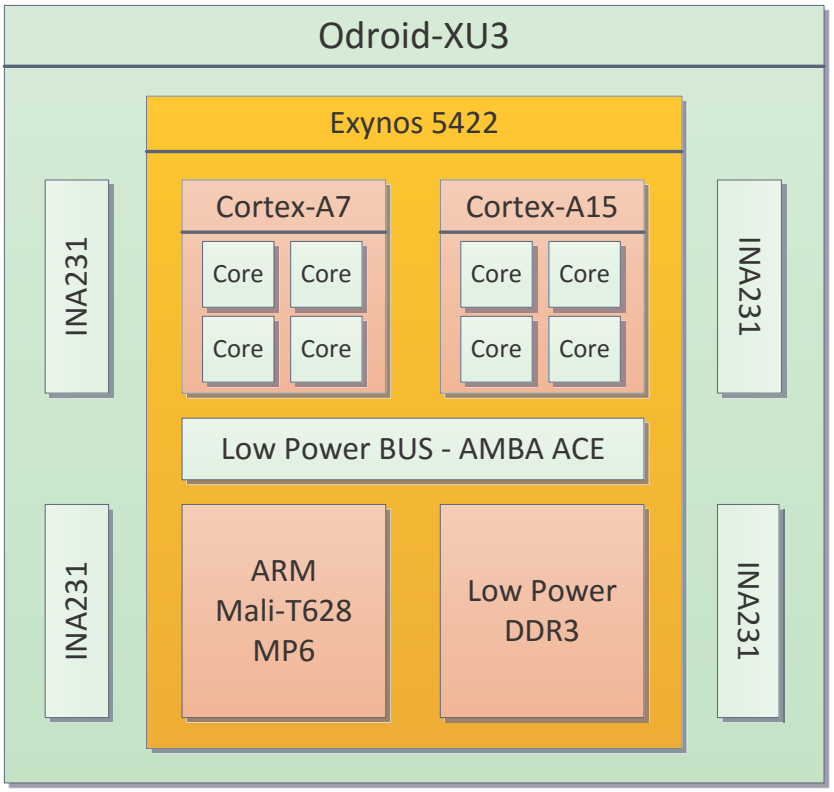


Figure 2.3: Odroid-XU3 structural overview.

1.1 profile support is integrated onto the chip enabling General Purpose Computation on Graphics Processing Unit (GPGPU). In addition, a shared 2GB low power DDR3 memory is available. All components are connected through an energy-efficient bus. Four INA231 [Tex13] sensors are used to measure current and power separately for "big" cores, "little" cores, the GPU, and the DDR3 memory. The sensors are connected via an I^2C serial bus to the MPSoC. A photo of the used board is shown in Figure 2.4. Already the prototype board is rather small, the physical dimension are $94mm \times 70mm \times 18mm$.

An Ubuntu Linux 14.04 LTS operating system, with a modified kernel based on kernel version 3.10.82, is deployed on the system. In contrast to the simulation-based system, this is a true multiprocessor operating system. The modifications we made to the kernel were necessary to enable a nearly seamless integration of the energy measurement and additional governors controlling the frequency of the processors. We created a tool to control the CPU's governors efficiently. Subsection 5.4.1.2 provides more details on how the governors are used. In addition, we developed two measurement applications, namely *EnergyMeter* and *Energy Relay Reader*. All tools and kernel modifications are available publicly at [SFB17b]. Section A.2 provides more details on the measurement software.

The Odroid-XU3 platform was used by several other researchers. For instance, Gensh et al. [GAR15] reported energy and performance results for various experiments. Further, Backes et al. [BRF15] considered the Odroid-XU3 as a platform for computer vision applications. The SLAMBench was evaluated on the Odroid-XU3 [NBZ15]. In addition, Prakash et al. [PWI15] exploited the system's heterogeneity for data parallel applications.

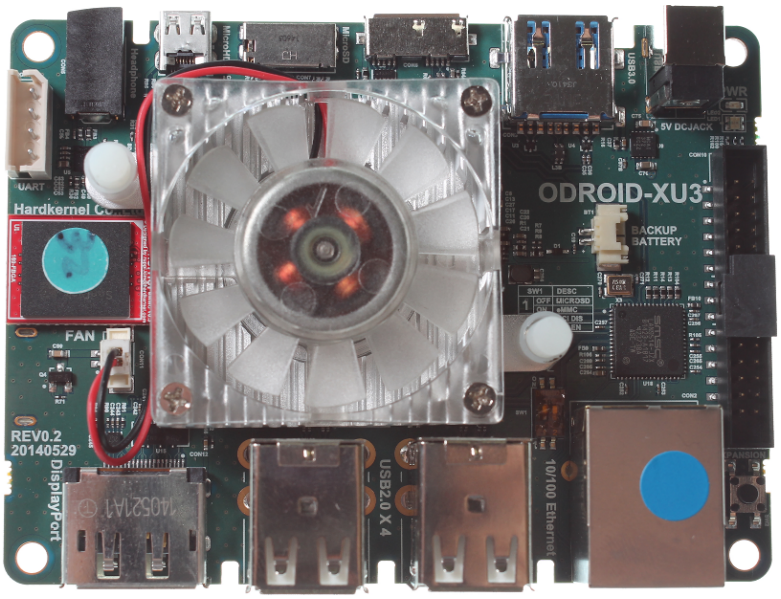


Figure 2.4: Odroid-XU3 development board.

They identified voltage-frequency scaling as an important parameter to achieve energy efficiency without large run time impacts. Aalsaud et al. [ASR16] used measurements results of the Odroid-XU3 to build an abstract system model.

2.2 Parallelism in Software

Traditionally, developers implement embedded software sequentially especially using low-level languages like C or assembler. To benefit from modern MPSoCs, developers must exploit parallelism in their software, regardless if they start with an existing sequential implementation or from scratch.

According to Asanovic et al., "Software is the main problem in bridging the gap between users and the parallel IT industry." Further, "Asanovic et al. experience teaching parallelism suggests that not every programmer is able to understand the nitty gritty of concurrent software and parallel hardware; difficult steps include locks, barriers, deadlocks, load balancing, scheduling, and memory consistency" [AWW09]. This thesis targets these challenges, starting from a sequential application, generating a parallel solution in a well-defined method. To start, we give a brief overview of the different types of parallelism typically used in software design, followed by a discussion of the challenges encountered during the parallelization of software, especially targeting embedded systems.

2.2.1 Types of Parallelism in Software

Whereas classical techniques to improve performance of sequential applications, like increasing the frequency of processors, reached thermal and power limits, parallel execution promises additional performance gains. Parallel execution is already heavily used in terms of bit-level and instruction-level parallelism. However, these fine-grained types

of parallelism are very limited and do not benefit from modern multiprocessor systems. In the following we present task-level, data-level and pipeline parallelism. These coarse-grained types of parallelism are key for exploiting the capabilities of modern parallel architectures, especially when following a sequential programming style or parallelizing sequential applications.

2.2.1.1 Task-Level Parallelism

Task-level parallelism partitions a program into sets of concurrent computations. On a very high abstraction level these tasks can provide (isolated) services like a web server serving multiple requests simultaneously. In this thesis we use a lower abstraction level for the definition of task-level parallelism considering concurrent execution of instructions and functions. Thus, this type is also often referred as functional parallelism describing the parallel execution of entire functions. Sometimes, tasks could be arranged in such a way that one task generates input for another task leading to a functional parallel pipeline. Benefits of functional parallelism include the fact that parallelism can be extracted from source code containing uncounted loops or complex data-dependent control flow inside the functions.

Figure 2.5 illustrates parallel processing on the same data without dependencies between the operations. Thus, the execution of the '+' and '-' computation can be executed in parallel leading to an asynchronous execution of the tasks. Figure 2.6 shows the parallel computation on different data.

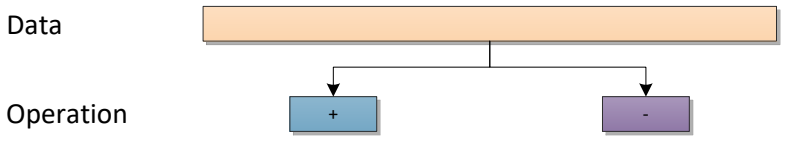


Figure 2.5: Task-level parallelism on same data.

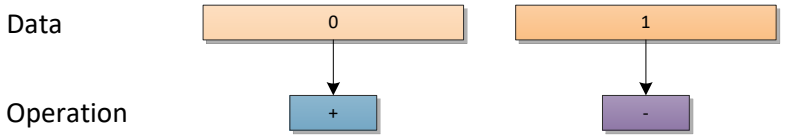


Figure 2.6: Task-level parallelism on independent data.

2.2.1.2 Data-Level Parallelism

The next type of parallelism splits a block of data into subsequences and processes the same operation on it in parallel. Figure 2.7 illustrates this paradigm. Here, the data is partitioned into four blocks which are then processed in parallel by the '+' operation. In case of GPUs, with thousands of compute units, data-level parallelism is the preferred paradigm. In extreme cases, each parallel instruction is executed synchronously across all processors.

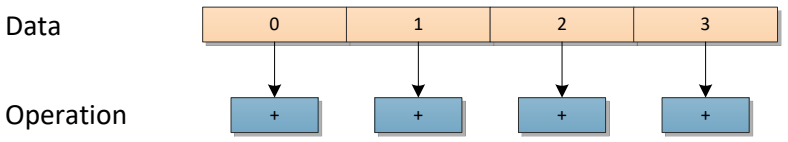


Figure 2.7: Data-level parallelism

The data-level paradigm is often applied to loops. For instance, a loop where each iteration manipulates a different element of an array will benefit from data-level parallelism. These iterations can be executed concurrently. Thus, this type of parallelism is also called *loop-level*, *doall* or *doacross* parallelism. Depending on the number of iterations mapped to a parallel task, loop-level parallelism is able to balance the computational load for target heterogeneous systems. A load balancing algorithm should take care that fast processors process a larger part of the iteration space than the slower ones.

2.2.1.3 Pipeline Parallelism

Pipeline parallelism has been used in fabrication for several decades where it's also called assembly line. Here, the result of one process stage is processed by a following stage. In case of software, a task produces the input of another task. Figure 2.8 visualizes an exemplary pipeline structure. As shown, the pipeline stages could work in parallel if more data is flowing through the pipeline. Pipeline parallelism is sometimes the only option to exploit parallelism in case of loop-carried dependencies. For instance, an image processing application applies a set of filters sequentially to a picture. This process can be pipelined if a set of pictures needs to be processed with an increased overall performance.

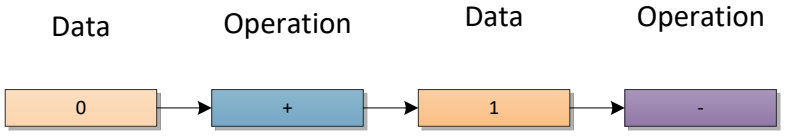


Figure 2.8: Pipeline parallelism.

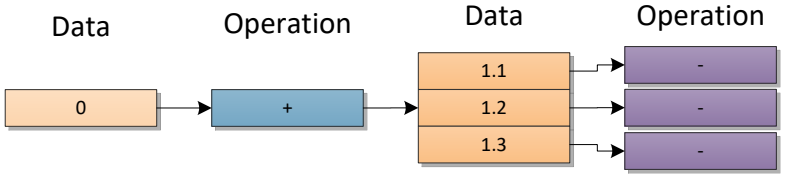


Figure 2.9: Hybrid pipeline and data-level parallelism.

To further increase the performance, processing stages can be duplicated to exploit nested data-level parallelism. Figure 2.9 shows this hybrid pipeline data-level parallelism. This paradigm can for instance, be applied to the previous image processing example. Here, each pipeline stage applies an operation like filtering. Some filters allow a parallel processing of pixels or blocks of pixels inside each image.

2.2.2 Challenges during the Parallelization

Creating parallel software is a time-consuming, complex and error-prone task. Especially, thinking in parallel dimensions is one of the major issues for software developer. For some problems, developers might fall back to existing known parallel algorithms, however, this is not always possible. Nowadays, a lot of effort is put into implementing parallel applications from scratch using high-level parallel languages. But, in embedded systems, applications are often designed sequentially as a sequence of consecutive steps including low-level languages to program special hardware components. In addition, existing legacy code must be reused to reduce the time-to-market. Thus, embedded software developers face the complex task to parallelize sequential applications.

In case of embedded systems, the used operating systems, if any, usually have no or only limited support for parallel applications. Thus, the developer must implement and control task creation, scheduling, synchronization etc. manually. One key design principle for resource-restricted embedded systems is to limit additional runtime overhead. Thus, decisions are usually taken offline at compile time, leading to a static parallelized application without or with just very limited dynamic behavior. This means that online load balancing or runtime mapping of tasks is usually not available and must be decided offline. Finally, "perhaps the biggest difference [...] is the traditional emphasis on real-time computing in embedded, where the computer and the program need to be just fast enough to meet the deadlines, and there is no benefit to running faster" [ACP06].

In the following we highlight four challenges software developers are facing during their parallelization work. Besides the question if a parallel implementation performs better, the developer first needs to identify regions in the applications benefiting from parallelism. Then, the developer must express the parallelism efficiently, dealing with data dependencies and efficient implementation for various target platforms.

Parallelism identification: Identifying regions in a sequential application which benefit from parallelization is a complex task. Several decades of research brought up manual, semi-automatic and automatic approaches. In the context of PA4RES, the PAXES tool [Cor13] identifies parallel regions in a sequential application.

Parallelism expression: After the identification of interesting regions the developer needs to express the parallelism. In the most complex way, the developer needs to rewrite the entire application and thus implement the extracted parallelism manually. This is obviously a complex and error-prone task. Especially keeping a good maintainability of the code is important. Thus, high-level approaches exist where the developer annotates for example loops which are then transformed by a compiler into a parallel application. This mostly keeps the sequential application and thus eases maintenance.

Dependency handling: During parallelization, the developer needs to deal with dependencies. Synchronization points and data dependencies must be identified. If

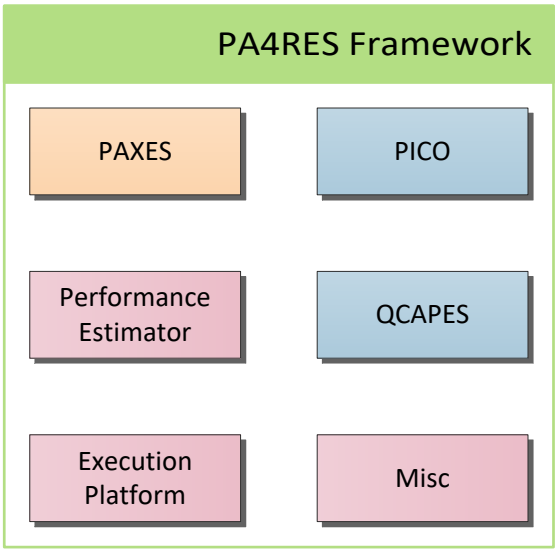


Figure 2.10: PA4RES framework overview.

data synchronization between concurrently running tasks is necessary, the developer needs to implement an efficient data exchange not wasting the benefits from parallel execution.

Efficient implementation: Finally, the developer must generate an efficient implementation of the application taking the characteristics of the target platform into account.

This overview highlights the complex task of creating parallel embedded applications utilizing modern embedded platforms. This thesis and the PA4RES approach provide solutions for these major challenges.

2.3 PA4RES - Framework

The PA4RES approach evolved from two European projects: MADNESS and MNEMEE. It combines the parallelization identification approaches developed in [Cor13] with the parallelization implementation and optimization techniques presented in this thesis. The PA4RES framework consists of a set of tools, libraries and miscellaneous files. Figure 2.10 illustrates major parts of the framework. Colors are used to distinguish work which is part of this thesis and work done in collaboration with others. Orange and gray are used for tools which are not developed by the author of this thesis, purple for tools partly developed by the author of this thesis and blue for tools entirely developed and designed by the author of this thesis. PAXES was developed in [Cor13], whereas the Performance Estimator (cf. Subsection 2.3.1), execution platforms (cf. Section 2.1) and miscellaneous parts have been partly developed or enhanced during this thesis. PICO and QCAPES were entirely developed during this thesis.

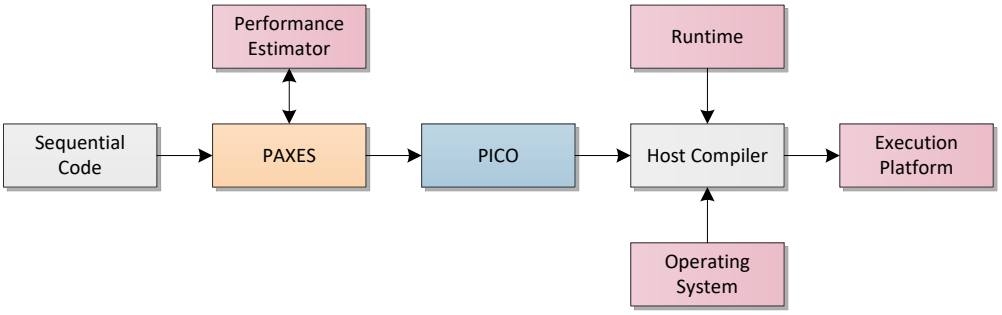


Figure 2.11: Classic parallelization tool flow.

All available tools of PA4RES are based on the MACCv2 [Pyk17] framework. This framework offers a standardized interface between tools and provides an abstract representation of the target architecture such as the amount of processors, the frequencies or information about the available memories. Using the MACCv2 framework, several tools can be connected to a tool flow. In the following, we present the classic parallelization flow, other tool flows will be introduced later in this thesis.

Figure 2.11 demonstrates the classic PA4RES tool flow to parallelize a sequential application. Sequential code is passed to PAXES in order to identify regions in the code benefiting from parallelization. Using smart algorithms in combination with the performance estimator, PAXES generates a parallel solution. In principle, this solution consists of the sequential source code instrumented with PICO annotations. Section 3.5 gives a detailed description of the supported annotations. Using MACCv2 internal interfaces, PICO takes the solution, analyzes the annotations and implements the parallel version with a source-to-source transformation process. After that, the generated source code is compiled by the target system's compiler and linked to a lightweight runtime library and the operating system. In case of the simulation-based systems, the RTEMS operating system is linked as a library. Finally, the executable can be executed on the target system.

2.3.1 Performance Estimator

Knowledge about the performance characteristics of the application is important during the optimization process. Detailed information like execution time or energy consumption of each individual instruction are key for sophisticated parallelization algorithms. There are several methods to obtain these data. The system manufacturer could provide this information, like an addition of two registers takes three cycles and consumes 5 pJ. With this information, an estimation algorithm could generate a model and perform a static performance estimation of the application. However, system manufacturers usually do not provide these data, at least not for the energy consumption, thus alternative approaches must be followed. One method is to build such models with extensive tests and reverse engineering. Here, each possible instruction and their combinations are executed and measured. Building such a generic model is a very complex and time-consuming task.

```
int a = 11;
int b = 22;
int c = a + b;
return c;
```

Listing 2.1: Example source code.

```
startRegion();
int a = 11;
stopRegion(); startRegion();
int b = 22;
stopRegion(); startRegion();
int c = a + b;
stopRegion(); startRegion();
return c;
stopRegion();
```

Listing 2.2: Instrumented source code.

```
startRegion();
int a = 11; // Similarly class 'assignment of constant'
stopRegion(); startRegion();
int c = a + a;
stopRegion(); startRegion();
return c;
stopRegion();
```

Listing 2.3: Instrumented source code with similary classes.

The PA4RES framework, with the Performance Estimator, follows a slightly different approach by generating such models dynamically. In principle, each C statement of the application is executed and measured on the target platform. Gathered performance values are then internally annotated to the C statements, accessible by all tools in the PA4RES framework. For that, the Performance Estimator instruments all instructions with appropriate *startRegion* and *stopRegion* function calls. These calls signal the target system, e.g. the simulator, to measure the run time and energy consumption of the statements between the calls. In the case of heterogeneous systems, the statements have to be executed on all processor types. Listing 2.1 shows an exemplary sequence of C statements passed to the Performance Estimator, Listing 2.2 lists the instrumented source code. The added function calls and the associated overhead can be significant. Therefore, the Performance Estimator supports the concept of *similarity classes* of statements to improve the run time of the analyzes:

Definition 2.2 (Similarity Class):
Two statements s and t belong to the same similarity class C if all introduction types used and their order in the statements are identical.

In a preprocessing step, the application is analyzed and similar statements are grouped into *similarity classes*. Then, only one representative statement of each class is analyzed. This effectively prunes the statements to be analyzed by trading precision of the generated model. Listing 2.3 shows that the algorithm detected that the two assignment statements are similar and thus only one statement needs to be evaluated. The Performance Estimator keeps track of dependencies and adjusts the statements, thus

they can be compiled correctly. Therefore, the Performance Estimator must ensure that the host compiler does not apply any optimizations which might remove the statements under test.

2.3.2 Parallelism Extraction for Embedded Systems

The Parallelism Extraction for Embedded Systems (PAXES) tool was developed by Daniel Cordes [Cor13] during his PhD thesis. Together with the author of this thesis, we published several papers [CEM12; CEN13b; CEN13a; CEN13c; Cor13] covering the different parallelization approaches included in PAXES. In this subsection, we will give a brief overview of the algorithms necessary to understand the philosophy driving the initial development of PICO and the extracted types of parallelism. Figure 2.12 highlights the important steps of the PAXES tool flow. PAXES starts the parallelization processes with a sequential application written in C. This source code is then transformed into an augmented hierarchical task graph. The augmentation includes performance estimations and data dependencies. The data dependencies result from a dynamic profiling run and thus might be imprecise. The task graph represents the hierarchical structure of the source code, for instance, a loop node has nested statement nodes for the loop body. This hierarchical structure is exploited in a bottom-up approach during the parallelization. PAXES provides Genetic Algorithm (GA) and Integer Linear Programming (ILP)-based algorithms to parallelize sequential applications. All approaches start at the innermost nodes and try to find parallel regions at this hierarchical level. According to the performance estimations and data dependencies, the sophisticated parallelization algorithms try to map statements to parallel tasks. PAXES thereby follows the idea that only one processor can execute one task at the moment. Thus, it limits the extracted parallelism to the number of available processors. Nevertheless, the user can enable extraction of more tasks than processors. After the current hierarchy level has been processed, the algorithms move a hierarchical level up and try to find parallelism at this level. Here it might be beneficial to combine parallelism found at this hierarchy level with parallelism found deeper in the hierarchies. At the top hierarchy level, the algorithms stop and the resulting solution is returned to the user. In case of multi-objective optimizations, a set of Pareto-optimal solutions is returned to the user who then selects the solution suiting the requirements best.

Basically, these solutions are the original sequential source code augmented with PICO annotations (cf. Section 3.5). PAXES uses a simplified data dependency detection and assumes that PICO identifies bad parallelization decisions. In addition, PAXES abstracts communication and only considers a fixed cost for the data exchange and thus might mispredict the performance impacts of the parallelization, as the evaluation in Section 4.5 revealed. Therefore, PICO is an essential part of the PAXES framework.

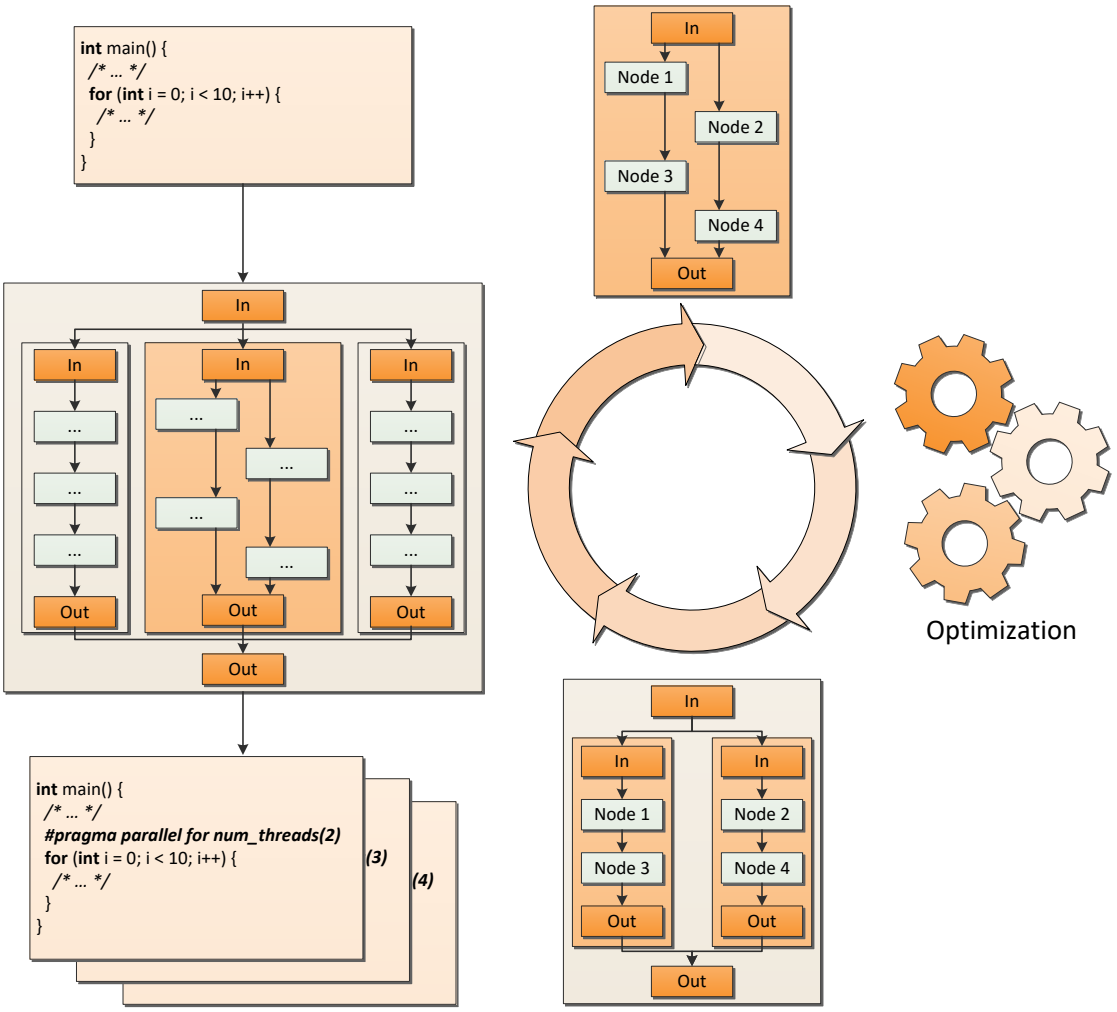


Figure 2.12: PAXES tool flow, adapted from [Cor13].

2.4 Conclusion

Parallelization offers additional performance gains not achievable with classical techniques like higher clock frequencies. In this thesis we focus on two experimental platform types, simulation-based and real hardware platforms using ARM processors. Both types are allocated in opposite sides of the performance spectrum. The simulator-based platforms represent low-power embedded systems with a limited operating system support, complex memory structure and heterogeneous processors. The real hardware platform uses a state of the art powerful ARM MPSoC used in millions of mobile phones. This system represents modern high-performance embedded systems, combined with a fairly powerful GPU enabling GPGPU computing on embedded systems. With versatile power sensing capabilities, this development board provides an interesting research platform. These platforms are represented using an abstract system model internally during the parallelization process.

Parallel applications are key to exploit the capabilities of modern MPSoCs. Task-level,

data-level and pipeline parallelism are the key paradigms used in this thesis. The identification of potential parallelism in sequential applications is already a very challenging task. But, expressing parallelism and dependencies efficiently is very important to support software developers facing the task to parallelize sequential embedded systems. The PA4RES framework offers a holistic approach to extract parallelism and implement the parallel application efficiently. Especially for the latter part, this thesis provides solutions, aiding software developers to create parallel applications running on low-power embedded systems efficiently.

Chapter 3

PICO-Framework

Contents

3.1 Introduction

The design of efficient MPSoC systems is a challenging task. Martin [Mar06] provides a comprehensive overview of challenges developers are facing. One key challenge is the efficient utilization of multiple processors. Programmers are used for decades to program their applications sequentially and a lot of legacy code should be reused for the new platforms. Manual implementation of parallel applications either from scratch or sequential legacy code is a complex, time-consuming and error-prone task, thus an automation of these processes or an easy code migration path is preferable. During the parallelization for embedded systems, bottlenecks like bad load balancing, bad resource allocation or using more parallelism than necessary must be avoided. These performance losses result from ignoring the heterogeneity as well as the resource limitations of embedded processors, memory systems, and communication channels. A good parallelization approach at the same time enables developers to reuse large libraries of existing C code. The PA4RES methodology aims to preserve the original sequential structure of existing application and therefore requires an approach that is able to create a parallel version from a sequential program with as little modifications as possible. Simple annotations should be used to express complex types of parallelism and hide complexity like the detection of data dependencies and their efficient synchronization. Further, this approach must support an automatic alignment of the implementation towards the capabilities, like distributed operating systems, of the target low-power system. This is key to run parallelized applications on such resource-restricted systems efficiently. To tackle these challenges and meet PA4RES' requirements, we developed the Parallelism Implementer and Communication Optimizer (PICO) framework. This chapter presents the parallelism implementation (PI) part of PICO whereas Chapter 4 focuses on the communication optimization (CO) part.

Today's existing parallelization techniques mostly target High-Performance Computing (HPC), which tends to be more homogeneous and at the same time much less resource-constrained compared to typical low-power embedded systems. As a consequence, using HPC tools is not an ideal solution to generate parallel software for constrained embedded systems and may waste optimization opportunities and result in overallocation of resources. A fundamental drawback of existing HPC approaches is that the underlying assumptions of given software structures do not fit to typical embedded applications, since embedded software may exhibit different control flow structures and data dependencies compared to high-performance computing. Whereas many HPC applications can be executed efficiently using coarse-grained task-level and fine-grained data-level parallelism originating from large data to process, many embedded applications only operate on small data sets or streaming data and thus exhibit more complex parallelization requirements.

This can lead to a substantial efficiency loss when applying HPC parallelization tools to embedded MPSoCs. Researchers developed several approaches to parallelize applications from the embedded systems domains. However, they do not satisfy the requirements of the PA4RES methodology. A detailed discussion of related approaches is given in Section 3.3.

In this thesis we developed the PICO methodology which provides a solution to efficiently create parallel applications for resource-constrained embedded MPSoC platforms running no, one or multiple real-time operating systems in a straightforward way. PICO is able to automatically detect data dependencies, implement necessary data synchronization and optimize the communication considering multiple objectives, like run time, energy consumption and memory requirements. Establishing such a system requires a large-scale effort, we did not start completely from scratch. Rather, PICO is in parts inspired by the well-known OpenMP [Ope17] syntax and semantics. We think this has a number of advantages since the basic parallelization model, API style, and annotation syntax are familiar to a broad range of developers. Thus, PICO perfectly suits for an iterative prototype driven parallelization process as well. However, our implementation is not derived from existing OpenMP variants. The major reason for this is that PICO tries to apply an established design principle of embedded systems, namely to accomplish as many decisions as possible at compile time, thereby reducing the required runtime overhead. As a consequence, decisions for scheduling and resource allocation are performed statically in PICO. Further, PICO does not require a shared memory architecture. Due to automatic dependency extraction, PICO can implement channel-based synchronization automatically. In addition, a port of OpenMP to a system with potentially multiple real-time operating systems with its limited hardware resources is a complex and not very promising task. However, the PICO approach inherits some features from OpenMP and we discuss the differences in Subsection 3.3.3.

This chapter covers work published in several forms. Initial results were presented in 2013[1]. A paper was published at SEPS [NEM14], leading to a journal article [NEM16]. In this thesis we provide detailed insights into PICO as well as extensions and updates to the existing publications. Section 3.2 introduces the application model and discusses requirements onto the input language. Related approaches are discussed in Section 3.3 and Section 3.4 describes the general structure of PICO. Section 3.5 presents the PICO annotations to express parallelism and Section 3.6 gives insights into the internals of the framework. The evaluation results are presented in Section 3.7 and Section 3.8 summarizes this chapter.

3.2 Application Model

The MNEMEE and MADNESS projects identified the *fork-join* execution model [Con63] as suitable for their requirements and PA4RES, as the spiritual successor, inherits this

[1] Olaf Neugebauer: "Design of an Infrastructure to Support Embedded Application Design for Heterogeneous MPSoCs (Research presentation)", M-SCOPES, 2013

model. In this model, a sequential task *forks* multiple concurrent tasks forming parallel regions. Inside these regions, new parallel tasks can be forked. At a subsequent point, the parallel tasks *join* back to a sequential execution. In our variant of the fork-join model, data exchange from sequential to parallel sections and vice versa as well as data exchange between parallel sections is possible.

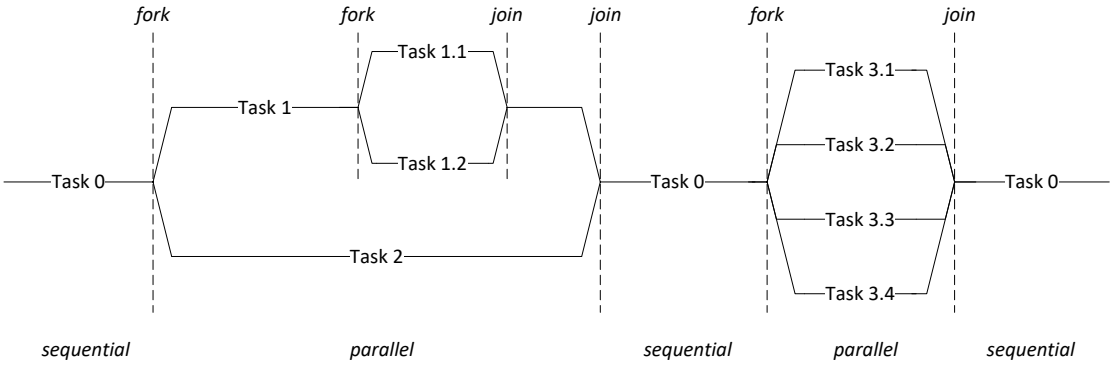


Figure 3.1: Fork-join execution model.

For clarification, Figure 3.1 illustrates an exemplary behavior of a parallel application following the fork-join model. Task 0 runs sequentially until the first fork point. Here, the flow splits into two parallel tasks and the original task suspends until the parallel region ends. Tasks 1 and 2 can now run in parallel and Task 1 also spawns two new tasks. Thus, three tasks can now be executed in parallel. After the parallel regions reach the join points, the sequential execution of Task 0 continues. At some later point, Task 0 spawns four new tasks for concurrent execution. Finally, all parallel tasks are joined and Task 0 resumes. In our model, this behavior could also be wrapped in a loop typical for embedded systems. For instance, a data stream is processed and for each element of the stream, the parallel behavior from Figure 3.1 is applied.

3.2.1 Communication Model

Data access is crucial in every application. Especially for parallel applications, data access and the accompanying synchronization are an essential, complex and error-prone job. In the PA4RES methodology, it is important that data is always consistent and coherent. Thus, PA4RES employs a hybrid communication model using sharing and message passing-based data exchange. During the parallelization process, the framework identifies data dependencies and integrates proper synchronization. Starting with a sequential application, PICO preserves the read and write order of data during the parallelization. PICO achieves this with dedicated FIFO channels to communicate data between concurrently running tasks. Here, data is *produced* by one task, transmitted through an exclusive channel and *consumed* by another task. This producer-consumer paradigm ensures that the data access order is equal in the sequential and parallel application. This approach is comparable to deterministic message-passing concurrency. In theory, with unlimited channel sizes, writing to a channel is always possible and

reading blocks if no data is available. However, in reality, physical limitations like available memory needs to be taken into account. PICO can map these channels either to (shared) memory or dedicated communication hardware and provision the channels to meet certain requirements like energy budgets. Our methodology allows a manual relaxation of the concurrency model. Thus, data can be stored unsynchronized in a shared-memory accessible by all processors globally if available. However, this can violate the order of memory accesses and produce wrong results. Private data should be stored in local memories to reduce the pressure on the memory system and communication infrastructure. Chapter 4 provides details on the communication optimization of PICO.

3.2.2 Structure and Components of the Application Model

Internally, PICO works on a graph representation of the application. This graph contains the following node types:

Statement Node: This node represents a statement. In the initial stages, these nodes are constructed from the sequential source code. In later stages, PICO might add additional statement nodes to model the parallel version correctly.

Fork Node: A fork node splits the (sequential) control flow into concurrent one. A fork node is connected to one or more task in nodes.

Task In Node: A task in node (*taskIn*) is the first node of a task and bundles the incoming control and data flow. A task always starts with a task in node.

Communication Out Node: A communication out (*comOut*) node models the explicit data synchronization between concurrently running tasks. Therefore, it knows the corresponding communication in node, the data to synchronize. To model the FIFO semantic, the communication out node also keeps track of the concrete channel parameters like mapping and buffer size.

Communication In Node: The commutation in (*comIn*) node models the receiving side of the channel-based synchronization. It is internally connected to the related communication out node to get access to stored information like the buffer size. This eases the maintainability since only the *comOut* nodes store the vital data.

Task Out Node: The task out node (*taskOut*) bundles the task internal control and data flow leaving the task. Therefore, a task is always exited at the task out node.

Join Node: The join node bundles the concurrent control flows, thus this node is directly connected to the task out nodes.

With these basic components we can define the PICO task as follows:

Definition 3.1 (PICO Task):
A PICO task starts with a task in node and ends with a task out node and may contain all other types of nodes.

Therefore, in the context of this thesis, we use the term task in the sense of a PICO task. Such a PICO task can be executed by an operating system thread or implemented directly for the target system. Beside the actual node type, the nodes themselves may contain the following additional information:

Task assignment: This field represents the assignment of the node to a specific task.

Iteration count: In case of counted loops, the nested nodes provide information on the iteration count.

Costs: Nodes may provide execution costs in terms of run time and energy consumption.

Dependent interations: In case of parallel loops, this field provides data indicating in which iteration number this node is executed.

Different types of control and data edges are used in the application model:

Control Edge: A control edge represents the normal control flow.

Loop In Edge: The loop in edge models the hierarchical structure of a loop in case of entering the loop.

Loop Back Edge: The loop back edge models the hierarchical structure of a loop in case of exiting the loop.

Call Edge: A call edge symbolizes a call to a function.

Return Edge: Leaving a function is modeled with a return edge.

Data Edge: A data edge models the data flow inside the application. Therefore, it models the dependency type and direction of the data flow.

Section 3.6 gives a detailed description of the internal processing steps to obtain such application model. There, the entire process starting from a sequential representation to this parallel model is presented. However, in the following, we give a brief example of the application model. Figure 3.2 shows a comprehensive visual representation of the application and communication model used in the PA4RES methodology. In this graph, the purple circles represent statement nodes, the orange triangles are used for fork and join nodes, light purple and orange nodes represent communication points, control flow is shown by solid black lines and data exchange/dependencies are drawn with red dashed lines. In this example, a sequential task forks two parallel tasks and passes data through a designated task in node to one child task (task on the left side). During the implementation, the framework decides with respect to the target platform if the data needs to be transmitted e.g. in case of distributed memories. Both parallel tasks have a nested loop. In this example, the computation node in the left parallel task produces data which is consumed in the other task. Communication nodes, representing FIFO channels, are used for this data exchange. Finally, the right parallel task produces data used in the sequential parent task. Section 3.6 gives more details on the actual internal implementation of this model and the necessary graph transformations.

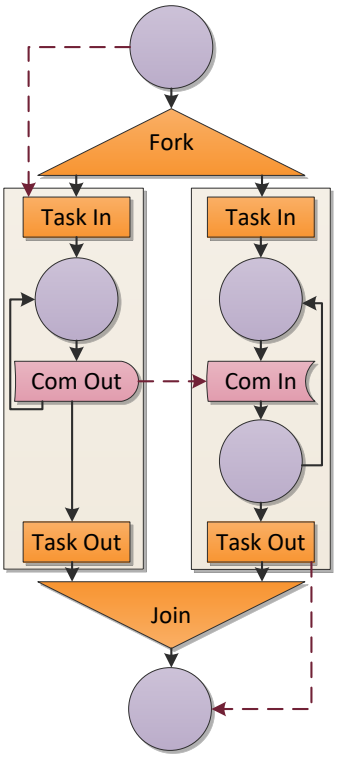


Figure 3.2: Abstract application model with communication used in PA4RES.

3.2.3 Programming Language Requirements - Parallelizable C

The PA4RES framework processes applications written in C, a widely used programming language, especially in the embedded systems domain. The language allows direct memory manipulation and low-level programming, often used to access dedicated hardware or to ensure timing predictable behavior. However, due to several side effects, like complex pointer arithmetic analysis, an automatic parallelization approach for C is quite challenging. Therefore, PA4RES exposes some requirements to the input source code. In general, the *Parallelizable C coding rules* [MOK10] are a good starting point. Beside these rules, PA4RES requires knowledge about the iteration space for the loops to be parallelized. This is necessary to realize a static scheduling, reducing the runtime overhead requested by the embedded system nature. This knowledge can be derived from fixed loop iteration counts. Bounds for simple loops can be analyzed and calculated by PA4RES automatically. For complex loops, *flow facts* [KKP11] annotations are required. However, the user can pass object files to the framework containing low-level programming or complex memory interaction. In such a case, PICO does not consider that object file and assumes it does not produce side effects.

3.3 Related Work

So far, researchers developed a number of tools and approaches to parallelize sequential applications. The first approaches we present target the implementation of parallelism

based on annotations in sequential application source code, especially focusing on methods targeting the embedded domain. The following approaches require more (destructive) modifications to the original source code. Finally, we present OpenMP related approaches. Some of the work discussed here will be revisited in Chapter 4 in perspective of the communication implementation and optimization properties.

3.3.1 General Overview

Generally, parallelizing compilers and automatic parallelization techniques have been in the focus of research for many decades (cf. [Mid12; BEN93; KA02]). Without a claim of completeness, a few important inspiring works must be mentioned. The SUIF parallelizing compiler suite [WFW94] set the foundation for many important publications like [HAA96] or [LDB99], highlighting the importance of human knowledge during the parallelization process. An automatic pipeline extraction with Decoupled Software Pipelining (DSWP) algorithm was presented by Ottoni et al. [ORS05]. Tournavitis et al. [TWF09] presented interesting approaches where they used machine learning to find candidate loops benefiting from parallelization. Then, OpenMP was used to parallelize those loops. Later [TF10] extended their work to pipeline parallelism. According to the PA4RES methodology, the parallelization proceeds in two phases. PAXES detects parallelism and PICO implements the parallelism. Therefore, we shift the focus to work which is more related to PICO.

IMEC's MPSoC Parallelization Assist (MPA) [IMM10; MBA09; BBW09] provides an approach to parallelize sequential applications especially targeting MPSoCs. MPA targets data-dominated applications from the signal processing domain. To parallelize an application, the user groups C statements into blocks naming them with unique labels. A separate *parspec* file is then used to manually specify the type of parallelization for the labels as well as shared variables and *LoopSyncs*, describing the interleaving iterations between data accesses in loops. For example, code block with label A should be parallelized with data-level parallelism. Using different *parspec* files, the user can explore several parallelization solutions without modification of the original source code. Communication is implemented via FIFO channels. The channel size is derived from manually specified *LoopSyncs* and is fixed in size and other implementation details. MPA was used in the MNEMEE project. In contrast to PICO, MPA does not consider energy consumption, memory requirements and limitations, as well as utilization of different memories for communication. Thus, MPA neglects optimization opportunities offered in the data exchanges of parallel running tasks.

The SPRINT tool [CDV07], MPA's predecessor, allows the implementation of functional parallelism specified by user directives. Thus, each task implements a different set of C statements grouped by C functions or labeled statement blocks. It is possible to map a single function, a single label or multiple consecutive labels to a task. Targeting streaming applications, the tool generates an executable concurrent SystemC model from sequential C code, consisting of FIFO-like communication channels and tasks. Similar to

MPA, the user specifies parallel functions or statement blocks and shared variables in a separate file. FIFO channels are inserted for all live variables crossing task boundaries. For each variable a separate FIFO channel is inserted. Thus, the resulting implementation is comparable to a Kahn Process Network (KPN). Selection of shared variables, and the size of the FIFO channels are done manually. Platform portability is achieved by leaving the implementation of the communication unspecified, instead SPRINT provides a set of standardized API calls. For each target platform, a library is necessary that implements the communication specified by the API. In contrast to SPRINT, PICO supports task-level, data-level and (hybrid) pipeline parallelism with respect to multiple objectives like run time or energy consumption.

An approach, more related to PAXES' task-level parallelization and the PA4RES approach in general, was presented by Ceng et al. in [CCS08] and later refined by Castrillón Mazo et al. [CL14]. A semi-automatic parallelization technique integrated in the MPSoC Application Programming Studio (MAPS) produces instrumented source code. MAPS uses the Tightly-Coupled-Thread framework (TCT) [ILK08] as a backend. TCT supports task-level and pipeline parallelism and inserts inter-processor communication automatically. *Thread scopes* defined by `THREAD` annotations are used to express the parallelism. Data exchange between concurrently running threads is implemented via message passing on dedicated communication hardware, provided by the custom-designed target hardware. In contrast to PICO, TCT focuses on homogeneous systems and therefore does not support load balancing of loops, especially important for heterogeneous architectures. Further, communication requires dedicated hardware modules. A later MAPS version proposes C for Process Networks [SSO14] allowing an application designer to describe parallelism manually through a KPN directly in C.

Thies et al. [TCA07] developed a parallelization approach leveraging coarse-grained pipeline parallelism in C applications. Using a set of annotations, the user expresses parallel loops with pipeline stage boundaries. Besides plain pipeline stages, this approach also supports the construction of hybrid pipeline stages. Communication is inserted automatically between the pipeline stages in case of stage boundary crossing data dependencies. Thies et al. identified static data dependency analysis for general C programs as a challenging task. Thus, this approach proposes an unsound dynamic data dependency analysis based on Valgrind[2]. Using profiling runs with test data, the approach records all data dependencies inside the loop and derives from these observations the necessary data to transmit between the pipeline stages. The authors argue that this unsafe analysis is acceptable in the domain of streaming applications since the loop behavior over the entire run time of the application is usually stable. The quality of the analysis surely depends on good input data during the initial profiling. In the paper, they conducted six case studies demonstrating the applicability of their approach for streaming applications. However, in contrast to PICO's static analysis, this dynamic approach requires good training data and it is unclear if it is applicable to a wider

[2] `http://valgrind.org/` - instrumentation framework for building dynamic analysis tools

range of application types. Their approach requires strong operating system support and relies on standard inter-process communication mechanisms offered by the target system. Further, this approach does not consider heterogeneity or resource-restrictions and only supports pipeline parallelism.

So far, all presented approaches try to preserve the sequential source code of the application and just use annotations to express parallelism. In principle, those annotated codes could also be compiled to a sequential application. Since this thesis also follows this idea, we only present some exemplary approaches leveraging this requirement, allowing strong modifications of the code.

SoC-C [RFG08] targets an efficient design of parallel applications for System-on-a-Chip (SoC) by introducing language extensions for C. Following channel-based decoupling, SoC-C generates parallel applications according to manually added communication primitives. SoC-C thereby extracts pipeline and task-level parallelism, but data-level parallelism is not supported. Parallel versions are extracted by partitioning the source code according to the synchronization barriers. In contrast to PICO, the user of SoC-C needs a deep knowledge of the data dependencies to correctly identify and place the channel synchronization barriers for the necessary data. Since this approach extends the language and adds new instructions, the code cannot be compiled without the SoC-C compiler.

Kwon et al. [KKJ08] developed a parallel-programming framework which requires the programmer to specify the application as common intermediate code (CIC), following the idea of a separation of algorithms and platform-dependent implementations. Due to this separation, the application code is retargetable. CIC consists of two parts, application or task code and an architecture description. The task code uses high-level non hardware dependent API calls to map code to accelerator hardware as well as to interact with the system like reading files. Parallel tasks exchange data either through channels or shared-memory, proper API calls need to be added by the user. Task-level and pipeline parallelism are specified using task definitions, data-level parallelism can be expressed using OpenMP directives inside these definitions. Besides hardware features, the architecture description also specifies task dependencies and constraints like real-time requirements, energy budgets or memory limitations. Finally, a CIC-translator translates the task code with respect to the architecture description into platform-dependent C code. Even if the application development approach differs from PICO's philosophy, using architecture knowledge to improve the performance and exploit the capabilities of the target system is also an integral part of the PA4RES methodology.

Verdoolaege et al. [VNS07] presented a technique which transforms sequential applications into KPN, which implicitly describes the parallelism of the application. Their approach requires the application to be specified as a (*Static*) Affine Nested Loop Program (SANLP). Unfortunately, all loops in the application have to be affine and consist of a single main loop which is not always the case in real-life applications. This approach is also used in the Deadalus framework [NTS08] and the MADNESS project.

3.3.2 OpenMP Related Work

OpenMP [Ope17] is the *de-facto* standard high-level shared memory programming model for desktop and high-performance computing. Its nondestructive pragma-based design made OpenMP attractive for developers. When limiting the used directives, the sequential solution is preserved and this eases the understanding and maintainability of the underling application. Traditionally, OpenMP compilers translates these directives into multithreaded applications utilizing runtime libraries. These special runtime libraries can be very heavyweight and usually do not exist for embedded systems. However, the accessibility of OpenMP encouraged researchers to investigate its applicability for embedded systems. In the following, we present related work, discussing OpenMP for embedded systems, pipeline parallelism and automatic dependency detection. Subsection 3.3.3 highlights differences between OpenMP and PICO as well as the historical the approaches.

The Multicore Association (MCA) [Mul17] provides a standardized API for communication, resource management and task management. Vendors and projects focusing on embedded multicores implemented these APIs, like for example EMB^2 [Sie17]. Wang et al. [WC13] developed libEOMP, a portable OpenMP runtime library focusing on embedded systems. This library maps OpenMP constructs onto the MCA APIs with a source-to-source translation implemented using the OpenUH [LH07] compiler. They demonstrated that libEOMP does not introduce significant overhead and the performance of applications developed with libEOMP perform comparable to vendor-specific approaches. Sun [SCZ15] later extended the work and presented a runtime library enabling a mapping of OpenMP *task* and *taskgroup* constructs to MCA APIs. The presented approaches show that OpenMP can be mapped to embedded systems, however, they still rely on the availability of the MCA libraries.

OpenMP has been ported to embedded systems by either relying on the available operating system [IAF12] or following a bare-metal approach such as Stotzer et al. [SJA13]. Using a vendor specific light-weight multi-core library called Open Event Machine, Stotzer et al. mapped OpenMP to a heterogeneous embedded system. In addition, they experimented with an early prototype implementation of OpenMP's acceleration extension, allowing offloading to DSPs. Further, they stress the importance of different memory management and cache coherence techniques used in heterogeneous systems in context of OpenMP's relaxed synchronization protocol. Jeun et al. [JH07] implemented OpenMP programs on MPSoC without an operating system support. Their approach supports just a subset of OpenMP directives. Chapmann et al. [CHB09] also identified the support of different memory architectures and offloading as a missing part in OpenMP. They introduce a *target* clause for OpenMP tasks to execute a task on a specific target processing unit. Their approach was implemented in a source-to-source translation using the OpenHU [LH07] compiler. Liu et al. [LC03] also propose extensions for OpenMP to enable work-sharing across multiple DSPs.

The research group of Luca Benini extensively investigated the usage of OpenMP in

the context of complex memory hierarchies typically found in embedded systems. Just to name a few, Marongiu et al. [MB09] proposed language extensions to OpenMP to allow programmers to map data to local memories, like SPM. Their approach is able to utilize explicitly managed memory hierarchies which are often found in embedded devices. They implemented their extensions in a custom compiler. Later, they extended their work with a compiler-based semi-automatic array partitioning [MB12] using profiling information. The framework assists the developer in distributing the memory access to different memory locations to improve the performance. Burgio et al. [BTM13] developed a runtime layer for OpenMP tasks targeting embedded shared memory clusters. A work-queue design is used to distribute the work. For such clusters, Marongiu et al. [MCT15] developed a programming model based on OpenMP. Their model is intentionally simple to increase the usability. For the target many-core systems, organized in a multicluster design, the presented offload directives achieved speedups comparable to hand-optimized OpenCL code.

Enhancing the execution model of OpenMP promises benefits of combining relaxed and sequential synchronization. Adding channel-based or message passing data exchange to OpenMP seems promising to bridge the gap between the shared and distributed memory world. Especially for the high-performance computing domain this hybrid model was examined by several researchers. For instance, Rabenseifner et al. [RHJ09] provide some cases where such a hybrid model led to significant reduction in data exchange, memory consumption and improved load balancing. Lusk and Chan [LC08] conducted experiments with a combined OpenMP and MPI [MPI17] approach. Pipeline parallelism might sometimes be the only option to parallelize an application. Stream programming allows to explore task-level, data-level and pipeline parallelism. Pop and Cohen [PC11; PC13] enriched OpenMP 3.0 with stream programming directives. By changing the execution model, pipeline parallelism can be naturally expressed with their extensions to OpenMP. Data dependencies between the pipeline stages are specified explicitly by the developer.

The presented work emphasizes the importance of memory hierarchies, especially different memory types as well as offloading or task mapping in general to exploit the capabilities of modern heterogeneous MPSoC. PICO is capable to explore different memories for data exchange and provides an intuitive way to map computation to different processing units. In addition, PICO does not rely on vendor specific library support and due to its source-to-source transformation PICO can easily be adapted to new platforms. PICO does not follow the user-defined synchronization, relaxed shared-memory model employed by OpenMP, it uses channel-based communication per default. Using special directives, PICO can be configured to follow OpenMP's model.

3.3.3 Distinction from OpenMP

Using OpenMP's syntax and semantics as the foundation for a parallelization tool for embedded systems is attractive due to its familiarity and its easy and seamless integration

into C code. However, in order to create efficient embedded parallel software especially targeting low-power systems, in the PA4RES project, we discovered that a number of additional approaches and extensions to the feature set of OpenMP were required.

A conventional OpenMP runtime environment is often not available for typical low-power embedded systems, for example, due to restricted APIs, employed process models or distributed operating systems. While porting an existing OpenMP implementation to a POSIX-compliant operating system is possible, this obvious approach has a number of drawbacks. Common OpenMP implementations rely on runtime decisions for task scheduling and additional resource allocations. For low-power embedded systems, in the PA4RES methodology, however, the amount of resources as well as the set of tasks are known at design time. This information are at least available after the parallelization extraction phase done by PAXES. Accordingly, most resource allocation and scheduling decisions can already be accomplished at compile time, which reduces the required runtime overhead significantly. This results in a very lightweight runtime library in contrast to OpenMP's standard library.

PICO provides a straightforward method to map tasks to specific processing units. In addition, offline load balancing of parallel loops is supported. Both aspects are important to efficiently utilize modern heterogeneous MPSoCs. OpenMP's *target* extension, however, focuses on offloading to accelerators like GPUs. Finally, one of the most relevant differences between HPC and embedded software is the more complex form of parallelism in embedded software. While task-level and simple data-level parallelism can be easily analyzed and annotated by an experienced programmer, pipeline parallelism and its dependencies are much less obvious and, consequently, much harder to detect and implement. Hence, PICO offers a well-defined method to express pipelines and provides data flow analysis enabling an automatic detection of data dependencies. Therefore, PICO provides automatic insertion of communication and synchronization primitives as required to guarantee a correct execution of the parallelized application. In order to adapt this to a large number of possible on- and off-chip communication infrastructures of different MPSoC, PICO not only supports communication over shared memories, but also using different communication infrastructures such as hardware FIFOs or NoCs. To achieve this, we would have to change the default synchronization model of OpenMP drastically, another reason we decided to develop PICO.

Initial work on PA4RES and PICO especially started in early 2012. In this time period, the OpenMP committee released first technical reports considering accelerators and coprocessor devices. With OpenMP 4.0 (end 2013), the support of heterogeneous systems was standardized. The OpenMP community discussed about support for pipeline parallelism around this time as well. Overall, PICO is inspired by OpenMP, but during the development time both approaches were not synchronized.

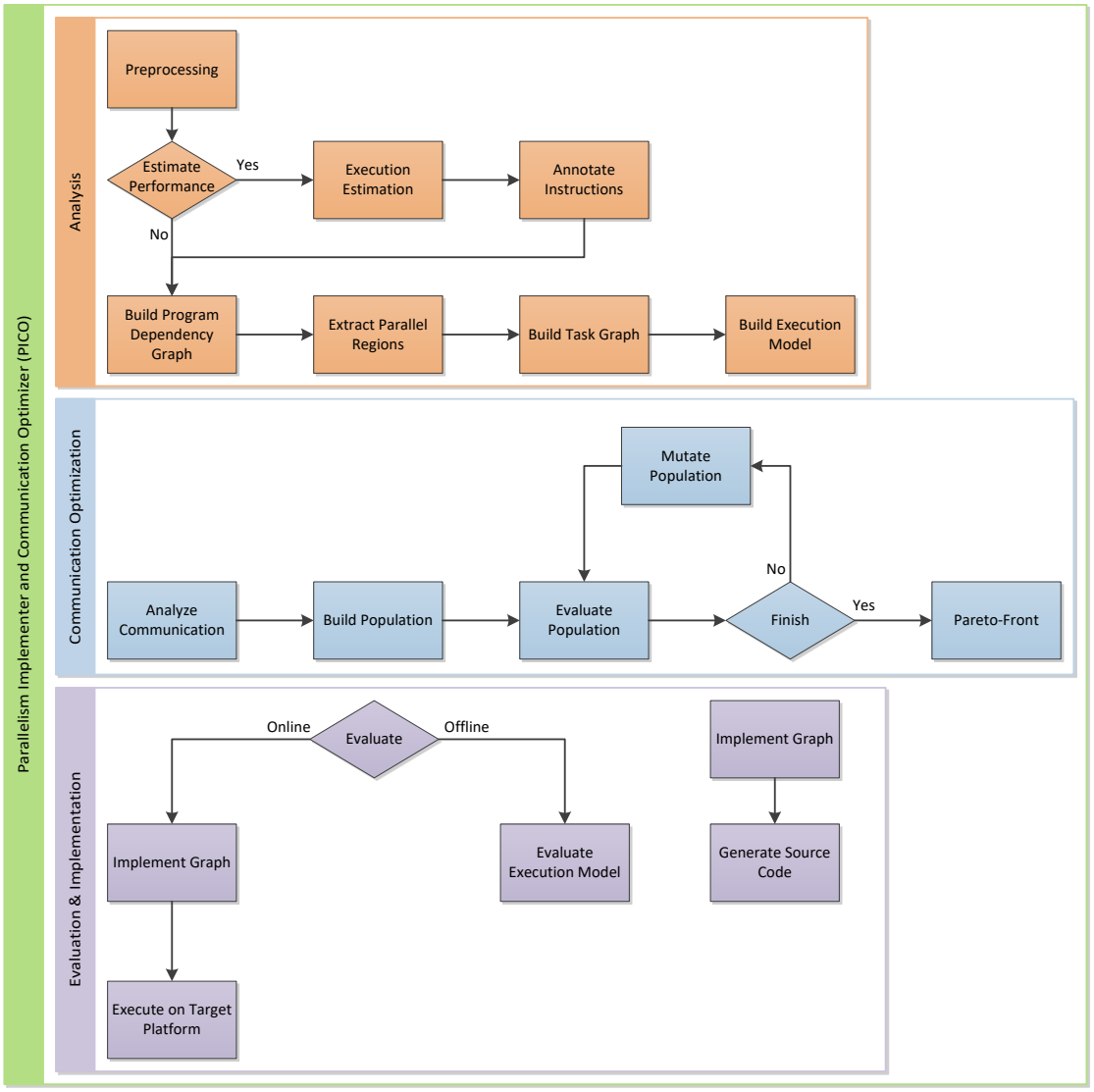


Figure 3.3: PICO framework overview.

3.4 PICO - Framework Overview

Figure 3.3 illustrates the internal structure of PICO. A flow diagram is used to highlight the internal flows and the major parts of the tool. Depending on the configuration, different parts are connected for a specified tool flow. PICO is split into three phases: analysis, communication optimization and evaluation and implementation. This chapter covers the analysis and implementation phases, whereas Chapter 4 is dedicated to the communication optimization. In the following, a brief overview is given and more details can be found in the related sections.

PICO is implemented in C++ following the MACCv2 methodology. Thus, PICO can interact with all available tools in the PA4RES framework through standardized interfaces. PICO uses the ICD-C compiler framework [Inf18] to generate a high-level abstract representation of the input code. This representation is called ICD-C Intermediate Representation (IR) and comparable to a collection of Abstract Syntax Tree (AST).

ICD-C also provides methods to manipulate the IR for code generation and basic data and control flow analysis.

In the analysis phase, PICO starts with annotated C source code (cf. Section 3.5) and transforms it into an IR. In a preprocessing step, PICO performs several optional program optimizations like constant and value propagation or constant folding with the intention to improve later analysis. These program transformations are partially supported by the integrated ICD-C compiler infrastructure. In an optional performance estimation process, PICO passes the (optimized) IR to the Performance Estimator. Resulting performance estimations are attached to the IR accessible by PICO. Using the IR, PICO then extracts the application model, representing the control and data flow of the input application. On this graph, PICO performs several graph modifications according to the selected parallelization strategies annotated by the user. The resulting task graph represents the parallel structure with added communication and task management. Section 3.6 provides more details on the internal processing steps. Finally, the analysis phase is also responsible for the construction of a simplified execution model of the parallel application. Subsection 4.4.2 shows how this model can be used during the communication optimization.

During the evaluation and implementation phase, depending on the selected flow, PICO emits the implemented parallel task graph as C source code and terminates or conducts a performance evaluation. In the latter case, the user chooses either an offline or online evaluation. The offline evaluation uses the previously constructed execution model. More details can be found in Subsection 4.4.2. In case of an online evaluation, PICO transforms the graph down to C code and then uses host compilers to generate executable binaries. It then performs evaluation runs and measures the performance of the application. Performance values, regardless of the origin (offline or online evaluation) can then be fed back to PICO's analysis phase, e.g. to steer the communication optimization process.

3.5 PICO Directives

This section presents the interface between the user, either a developer or PAXES, and PICO. As discussed above, PICO uses annotations inspired by OpenMP to express parallelism. We expect that the usability and acceptance of PICO will benefit by adopting this style, especially for new users. Like OpenMP, PICO uses `#pragma` directives to annotate parallel regions. The `#pragma pico` directive marks regions where PICO should perform the parallelization. To increase accessibility, OpenMP's `#pragma omp` directive can also be used. Resulting parallel programs use the fork-join application model presented in Section 3.2. An annotation consists of a *directive* describing the type of parallelism, whereas optional *clauses* add additional information and configurations. Reusing clause idioms in multiple directives eases the usage of PICO. The following section explains the available directives and clauses.

Parallelization directives partition the sequential source code into parallel regions. Such a region starts with the keyword `pico parallel` followed by the type of parallelism. In a parallel region, multiple tasks are specified which can be executed concurrently. In the PA4RES terminology, a task is a set of C statements (cf. Section 3.2). Tasks, for instance, can be executed by operating system threads. According to the PA4RES methodology, the sequential structure of the application code must be preserved. Thus, source code modifications, especially rearrangement of instructions, are not allowed. In our opinion, this eliminates possible pitfalls and thus quickly leads to parallel prototypes. To group statements to tasks, a clause named `taskid` is used. Otherwise, each parallel section will result in a new task. This enables the developer to assign statements to tasks freely without reordering them. The `num_threads` clause limits the number of parallel tasks. If the number of tasks is not specified by the developer, PICO generates as many tasks as available processing elements. We will use processor as a synonym for processing units. To designate a task to a specific processor, PICO provides the `processor` clause. The processors are identified with a unique id given by the MACCv2 framework. It is also possible to assign a task to a set of processors and PICO takes care of the final mapping. If no processor mapping is specified, PICO uses a heuristic to assign tasks to processors. Here, the algorithm takes the system capabilities reported by the MACCv2 framework into account. According to the employed data analysis, PICO implements communication to synchronize data between concurrent tasks. Since the implemented static C data analyses tend to generate conservative results, the user may specify data as `global` such that PICO does not implement data synchronization for this data.

3.5.1 Task-Level Parallelism

The `pico parallel sections` directive expresses regions employing task-level parallelism. The `pico section` pragmas partition the code inside a region into concurrent tasks. Listing 3.1 shows a parallel section with two nested tasks. In this example, three statements are mapped to two different tasks distinguished by numeric identifiers. Task 1 will be executed either on processor `1` or `3`. The mapping process later decides which processor should be used. In this example task 2 is allocated to processor `2`. Figure 3.4 shows a possible runtime behavior of this example.

```
#pragma pico parallel sections num_threads(2)
{
  #pragma pico section taskid=1 processor={1,3}
  a = b + c;
  #pragma pico section taskid=2 processor={2}
  d = func(b, c);
  #pragma pico section taskid=1 processor={1,3}
  e = a + b * 42;
}
```

Listing 3.1: Parallel sections.

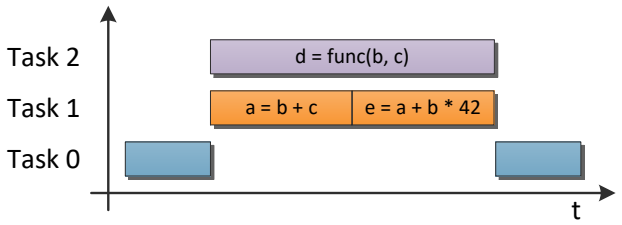


Figure 3.4: Runtime behavior: Parallel sections, cf. Listing 3.1.

3.5.2 Data-Level Parallelism

PICO provides the `pico parallel for` directive to parallelize counted loops. This method executes a loop as multiple concurrent tasks. According to the PA4RES methodology, iterations are scheduled statically and thus the allocation must be done during compile time. The developer can control the mapping behavior with special clauses. If no scheduling is defined by the user, PICO uses heuristics to generate a valid iteration schedule. PICO uses either a simple round robin mapping of iterations or a processor-dependent mapping. In the latter case, PICO assigns more iterations to the processors with a higher performance reported by the MACCv2 infrastructure.

```
#pragma pico parallel for num_threads(2)
for (int i = 0; i < 10; i++)
{
  a[i] = b * i;
}
```

Listing 3.2: Data-level parallelism - pico parallel for.

Listing 3.2 shows a loop that in each iteration modifies a different array position (memory location). Such loops offer the opportunity to exploit data-level parallelism. In this example, we instruct PICO to generate a parallel version of this loop with two tasks. In the case of a homogeneous system, PICO divides the total iteration space by the number of tasks and maps these consecutive iteration blocks to them. Figure 3.5 shows a possible runtime behavior.

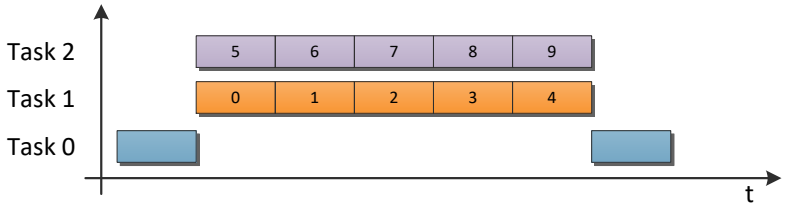


Figure 3.5: Runtime behavior: Parallel for, cf. Listing 3.2.

A balanced iteration mapping might not always be ideal. Thus, the developer can assign continuous iteration blocks to processors using the `chunks` clauses. A comma separated list of blocks is then allocated to the processors in a round robin way. For instance, the developer likes to map 20 iterations. Task 1 processes the first four iterations,

then task 2 a block of 10 iterations and then task 1 the remaining six iterations. For this behavior, the required clause looks like this: `chunks = 4, 10, 6`. Either the accumulated block sizes must equal the total number of iterations or the `interleaved` clause must be used. The `interleaved` clause instructs PICO to continue the previously defined chunk mapping until all iterations are assigned to tasks. Listing 3.3 demonstrates the use of the `chunk` and `interleaved` clauses. In this example, two tasks should be generated, the first executes iteration blocks with three iterations. The second tasks processes iteration blocks of size two. Figure 3.6 illustrates the runtime behavior of the `interleaved` clause. Task 1 executes iterations 0-2, 5-7 and task 2: 3, 4, 8, 9. Such static load balancing pays off for heterogeneous target platforms.

```
#pragma pico parallel for num_threads(2) chunks = 3,2 interleaved
for (int i = 0; i < 10; i++)
{
  a[i] = b * i;
}
```

Listing 3.3: Parallel for with chunks clause.

Let us assume that processor 1 and 2 are faster than processor 3. In Listing 3.4 we instruct PICO to map the large iteration block to either processor 1 or 2 and the smaller block to the slower processor 3. Figure 3.7 shows how beneficial this load balancing can be.

```
#pragma pico parallel for num_threads(2) chunks = 3,2 interleaved
    ↪ processor = {1,2}, {3}
for (int i = 0; i < 10; i++)
{
  a[i] = b * i;
}
```

Listing 3.4: Parallel for with processor assignment.

PICO provides an even more fine-grained method to map iterations to tasks. With the `iterations` clause, individual iterations can be assigned. This seems complicated for human users of PICO but PAXES can generate such fine-grained iteration mapping. Listing 3.5 shows such precise iteration mapping to two tasks and Figure 3.8 visualizes the resulting runtime behavior. To simplify, the developer can use "`...`" in the `iterations` clause to define a range of continuous iterations.

```
#pragma pico parallel for num_threads(2)
    ↪ iterations = {0,2,3,7,8}, {1,4,...,6,9}
for (int i = 0; i < 10; i++)
{
  a[i] = b * i;
}
```

Listing 3.5: Parallel for with precise iteration mapping.

The examples provided in this subsection may result in a slow parallel execution if a conservative imprecise data analysis does not detect that each loop modifies a different part of the array (memory) and thus an extensive data synchronization is not necessary. The developer can prevent PICO to add synchronization by using the `global` clause with the `parallel for` directive.

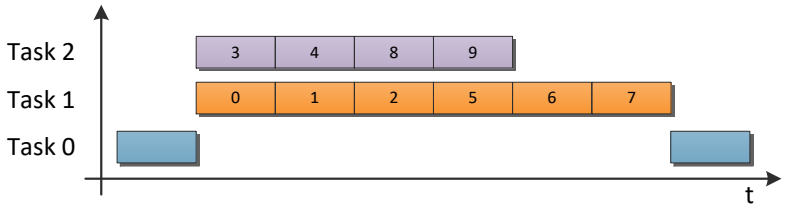


Figure 3.6: Timing behavior: Parallel for, cf. Listing 3.3.

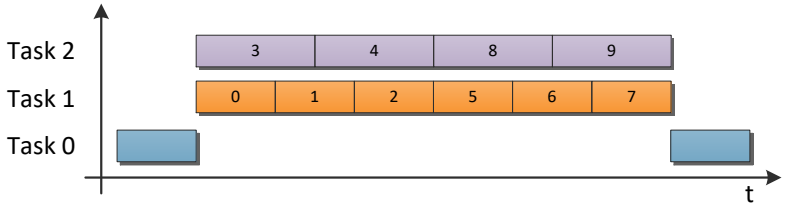


Figure 3.7: Runtime behavior: Parallel for, cf. Listing 3.4.

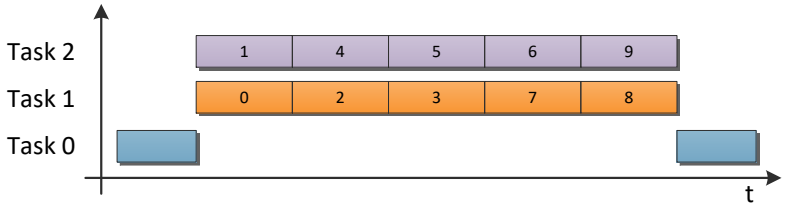


Figure 3.8: Runtime behavior: Parallel for, precise iteration mapping, cf. Listing 3.5.

3.5.3 Pipeline Parallelism

Pipeline parallelism might be the only solution to parallelize a loop with loop-carried dependencies. Listing 3.6 shows the main computation loop of the `Spectral` analysis benchmark from the UTDSP benchmark suite [Lee17]. This is a representative embedded application that calculates a power spectrum of an input speech sample. The `Spectral` implementation suffers from loop-carried dependencies preventing the utilization of data-level parallelism.

In detail, the outer loop contains two inner loops and a call to a *fft* (Fast Fourier Transform) function in between both loops. The second inner loop contains a loop-carried dependency to its previous iteration because it reads `mag[j]` which was written in the previous iteration of the outer loop. Due to this dependency, data or task-level parallelization of this loop is not beneficial.

```
for (int i = 0; i < 16; ++i) {
  int index = i*4;
  float sample_real[64];
  float sample_imag[64];

  for (int j = 0; j < 64; ++j) {
    sample_real[j] = input_signal[(index+j)] * hamming[j];
    sample_imag[j] = zero;
  }

  fft(sample_real, sample_imag);

  for (int j = 0;  j < 64; ++j) {
    mag[j] = mag[j] + (sample_real[j] * sample_real[j]
                    + sample_imag[j] * sample_imag[j]) / 4096;
  }
}
```

Listing 3.6: Sequential main loop of the Spectral analysis benchmark.

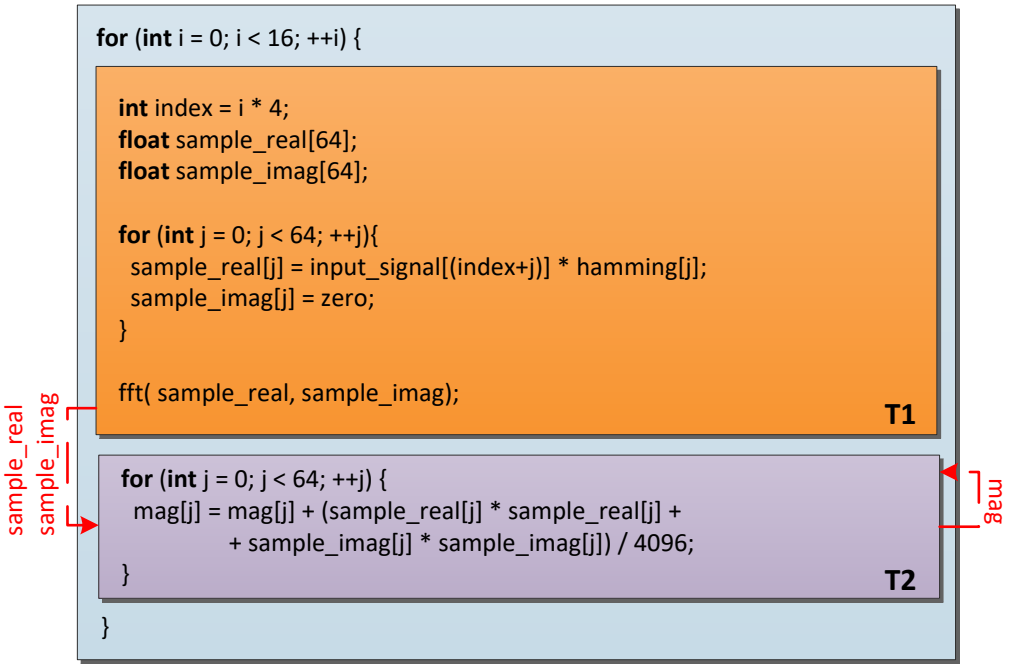


Figure 3.9: Main loop of Spectral analysis with pipeline parallelism.

Fortunately, the loop can be split into two stages using pipeline parallelism. Figure 3.9 shows the pipeline structure. Two tasks are generated from the sequential loop and executed in parallel if all data dependencies are fulfilled. The runtime behavior is illustrated in Figure 3.10. This graph highlights that Task 2 runs concurrently as soon as the data has been generated in Task 1.

To express such complex parallelism, PICO provides the `pico parallel pipeline for` directive. The directive is added to the loop in the example code and the pipeline

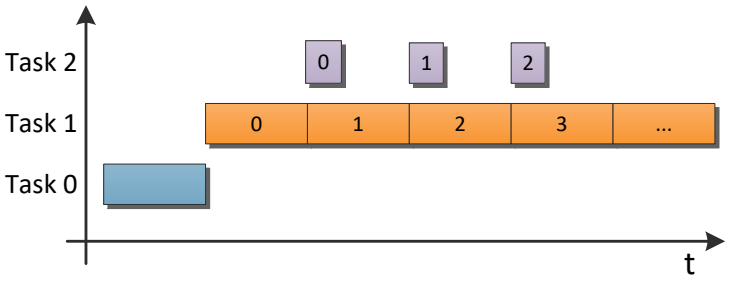


Figure 3.10: Runtime behavior for Spectral analysis with pipeline parallelism.

stages are specified using `pico section` directives. Similar to the `section` clause from Subsection 3.5.1, `taskid` can be used to group sections to tasks. Listing 3.8 demonstrates that only a few annotations are required to express this rather complicated parallelism. During the implementation, PICO automatically takes care that the data from task T1 is forwarded to task T2.

```
#pragma pico parallel pipeline for num_threads(2)
for (int i = 0; i < 16; ++i) {
  int index = i*4;
  float sample_real[64];
  float sample_imag[64];

  #pragma pico section taskid=1
  {
    // for loop + fft()
  }

  #pragma pico section taskid=2
  {
    // for loop
  }
}
```

Listing 3.7: Parallelized main loop of the Spectral analysis benchmark.

3.5.4 Hybrid Pipeline Parallelism

With pipeline parallelism, the main computation loop of the `Spectral` benchmark can be parallelized successfully, but there is more parallelism nested. As described in Subsection 2.2.1.3, additional data-level parallelism can be extracted from pipeline stages. Figure 3.11 illustrates how the first pipeline stage is duplicated and thus the workload is distributed to two additional tasks. In this case, PICO ensures the data transfer from these three tasks to the second pipeline stage. In the second pipeline stage, PICO takes care in which order the communication channels are read such that the parallel behavior of this loop is equal to the sequential solution and to prevent deadlocks. Figure 3.12(a) depicts the runtime behavior of this hybrid pipeline.

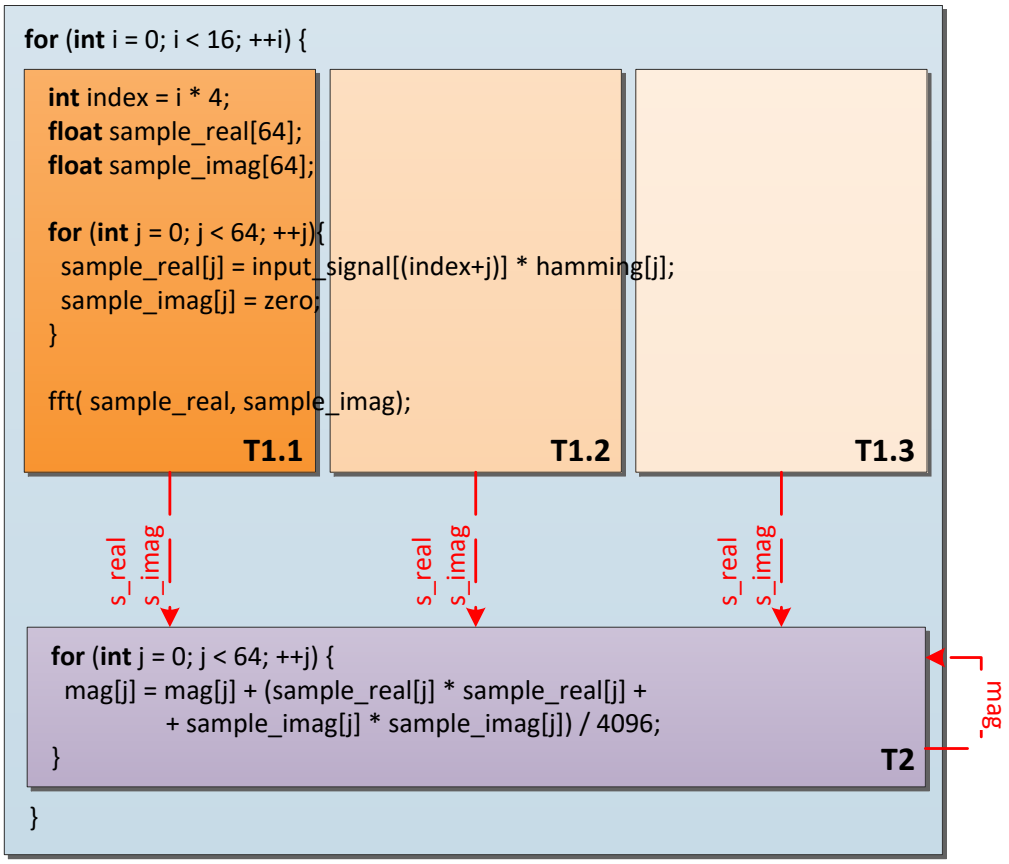
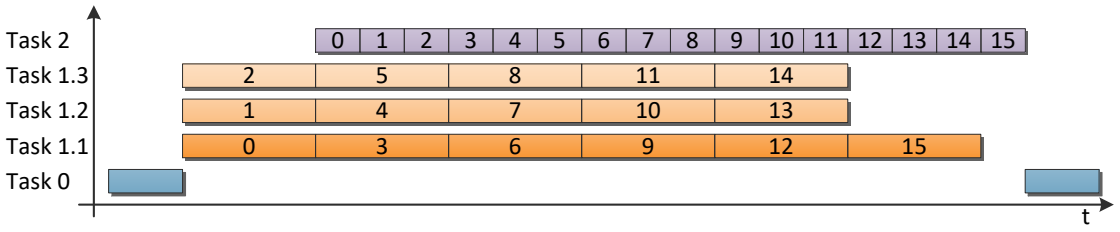


Figure 3.11: Spectral analysis with hybrid pipeline parallelism.

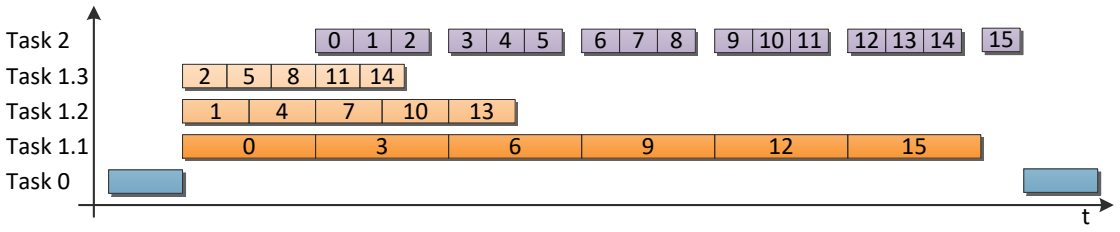
```
#pragma pico parallel pipeline for num_threads(4)
for (int i = 0; i < 16; ++i) {
  int index = i*4;
  float sample_real[64];
  float sample_imag[64];

  #pragma pico section taskid=1 chunks = 1,1,1 interleaved
  {
    // for loop ...
  }

  #pragma pico section taskid=2
  {
    // for loop
  }
}
```

Listing 3.8: Parallelized main function of the Spectral analysis benchmark.

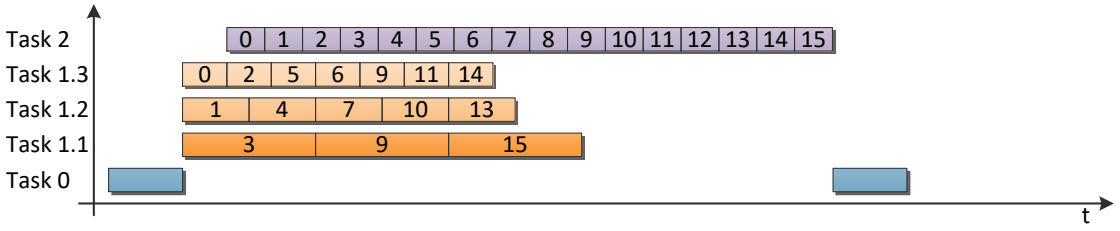
PICO supports `chunks`, `interleaved` and `iterations` clauses for hybrid pipeline stages. To increase the usability of PICO, these clauses work similar to those used for data-level parallelism presented in Subsection 3.5.2. Here, each task executes iteration blocks of size one, as seen in 3.12(a). The `taskid` clause is not necessary in this example



(a) Homogeneous system



(b) Heterogeneous system, unbalanced



(c) Heterogeneous system, balanced

Figure 3.12: Runtime behavior for Spectral analysis with hybrid pipeline parallelism.

since this pipeline stage only consists of continuous statements. Executing this example application on a heterogeneous system might result in a poor runtime behavior like shown in 3.12(b). Therefore, `processor` clauses and varying chunk sizes or precise iteration mapping should be used to statically balance the iterations according to the capabilities of the target architecture. 3.12(c) shows the improved behavior after applying PICO's powerful load balancing modifications.

3.6 Internals of PICO

This section presents details of the internal processing steps of PICO. Section 3.2 discusses the overall application model, communication model and requirements to the input languages. The entire PICO core framework results in roughly 20.000 C++ Source Lines Of Code (SLOC) and additional 1.000 SLOC for the RTEMS and Odroid runtime libraries. Test and evaluation scripts, mostly in Bash and Python, accumulate up to 2.000 SLOC. Applications and synthetic test cases are part of PA4RES, however, the author of this thesis contributed significantly to the application repository.

Embedded in the PA4RES project, PICO utilizes the MACCv2 framework. It can be executed independently or as part of the entire tool flow. The result of PICO's source-to-source transformation is a set of source code files, containing all the necessary

code for the parallel execution. At this stage, API details are intentionally left unspecified. Hence, the runtime library takes care of the platform-dependent implementation. This combines portability and optimized target specific implementations.

PICO provides detailed information to the user regarding the transformations and implementation. A comprehensive summary contains, for instance, a list of necessary data synchronization with variable names and channel details, number of tasks and their mapping. Besides the textual summary, PICO also provides graphical feedback to the user. All relevant internal graphs can be exported for a manual inspection using the GraphML [Gra17] standard. In this thesis, we use yEd from yWorks [yWo17] to generate the graphical representation of the GraphML files. In the following section we present the analysis and implementation phases in detail.

3.6.1 Analysis Phase

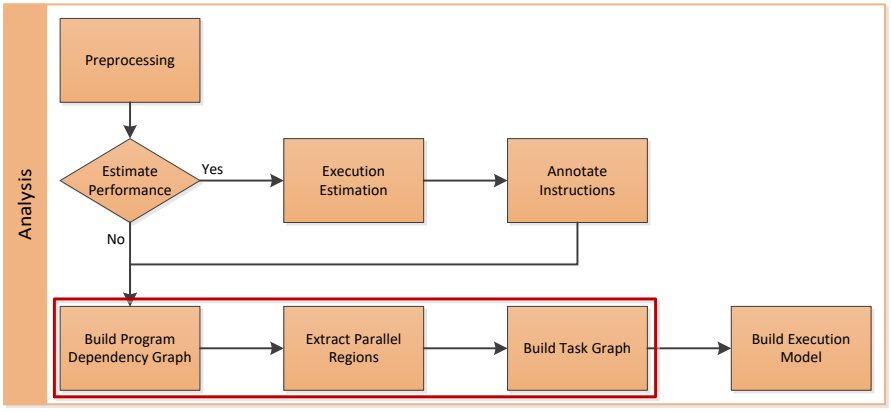


Figure 3.13: Analysis phase of PICO.

The analysis phase transfers the plain C source code into a graph representation suitable for the parallelization. Figure 3.13 shows the main analysis components and their interaction. In the following, we focus on the three highlighted components, the construction of the Program Dependence Graph (PDG), the extraction of the parallel regions and finally the transformation from the PDG to the task graph which correspondents to the application model (cf. Section 3.2). Therefore, we introduce some compiler fundamentals to ease the understanding of the analysis phase. The following definitions are based on standard compiler literature [ALS06; Muc97; KA02; App97]. In general, an application can be defined as a sequence of instructions, statements or basic blocks:

Definition 3.2 (Basic block):
A basic block is a sequence of instructions or statements which is always entered at the beginning and exited at the end of the sequence.

In this thesis we assume that a graph is always a directed graph:

Definition 3.3 (Directed Graph):
A directed graph $G = (V, E)$ is an ordered pair of nodes $n \in V$ and edges $(n_i, n_j) \in E$, $E \subseteq V^2$. A directed edge connects two nodes $n_i, n_j \in V$ with an explicit relation, (n_i, n_j) represents a directed edge from n_i to n_j. A graph might contain cycles, a directed graph without cycles is a Directed Acyclic Graph (DAG).

In cases where the direction of edges is irrelevant, we emphasize this fact. Each node in a graph has a set of *successor* and *predecessor* nodes:

Definition 3.4 (Successor, Predecessor):
Let $G = (V, E)$ be a directed graph. For a node $n \in V$, the set of predecessor nodes is defined as $pred(n) := \{u|(u, n) \in E\}$ and the successors $succ(n) := \{v|(n, v) \in E\}$.

A graph, representing all possible sequences of instructions in an application is a Control Flow Graph (CFG):

Definition 3.5 (Control Flow Graph):
A Control Flow Graph (CFG) is a directed graph representing the execution flow in a program. A node can represent a single instruction, complex statement or basic block. Edges represent the execution order of the nodes. A intraprocedural CFG represents the flow inside a function whereas a interprocedural CFG models the flow across function boundaries.

A call graph expresses the relation between functions:

Definition 3.6 (Call Graph):
A call graph is a CFG with nodes representing functions and directed edges representing the caller-callee relations.

Data dependencies between nodes must be modeled:

Definition 3.7 (Data dependency):
A node x has a flow dependency or Read-After-Write (RAW) dependency to node y if x writes data which is used in y. x has an anti-dependency or Write-After-Read (WAR) dependency to y if y writes data which is read in x. Nodes x and y have an output dependency (Write-After-Write (WAW) dependency) if both nodes write to the same variable.

The collection of all data dependencies and nodes is defined as:

Definition 3.8 (Data Flow Graph):
A Data Flow Graph (DFG) is a directed graph representing the data dependency and thus the flow of data between nodes. Nodes are typically instructions, statements or basic blocks. Different edge types are used to distinguish between the dependency types.

PICO works on a Program Dependence Graph during the parallelization process:

Algorithm 3.1 Program Dependence Graph Construction

```
procedure constructPDG(IR ir)
    intraCFG ← constructIntraCFG(ir)
    callGraph ← constructCallGraph(ir)
    cfg ← constructCFG(intraCFG, callGraph)
    dfg ← calculateDataFlow(cfg)
    pdg ← constructPDG(cfg, dfg)
    return pdg
```

Definition 3.9 (Program Dependence Graph):
A Program Dependence Graph (PDG) is a directed graph with nodes representing statements, instructions or basic blocks and edges representing dependencies between the nodes. A dependency is either data or control-related. Thus, a PDG combines control and data flow into a single representation.

PICO uses custom data structures and algorithms for the graph processing. Several lookup data structures are used to enable a fast access to specific nodes, e.g. all nodes of a task. In addition, several graph transformation operations are supported e.g. node duplication or moving nodes in the graph. The graph implementation also provides a method to calculate distances or relations between nodes. PICO checks the consistency of the graph after each major transformation and thus ensures a correct parallelization. In the following section, we present how PICO constructs these graphs in more detail.

3.6.1.1 Program Dependence Graph Construction

The analysis phase starts with plain C code and uses the ICD-C to generate the IR. This step ignores annotated PICO parallelization directives. The IR basically represents all statements in the code similar to a collection of ASTs. Algorithm 3.1 provides a comprehensive algorithmic description of the construction process. PICO connects the AST into an intra-CFG for each function. According to the application model (cf. Section 3.2), a node represents a C statement and edges model the control flow. Programs may contain loops, special edges are used to denote entering and exiting of a loop. These edges are called *loop* and *back* edges respectively. Using ICD-C's call analysis, PICO then connects the intra-CFG to a inter-CFG by adding *call* and *return* edges.

Listing 3.9 shows a simple program with a loop containing a nested call. Figure 3.14 depicts the resulting inter-CFG generated from PICO's GraphML output. Nodes represent IR statements. It is obvious that ICD-C IR is closely related to the original source code. Each node has a unique identifier and a statement (data) assigned. The first and last nodes are colored red and normal statement nodes in green. Control flow is visualized with directed edges. Node 10 is a loophead statement, in the case of static iteration counts or annotated flow facts, the number of iterations is added. From that node, a *loop* edge connects to Node 7 and a *back* edge from 7 to 10 drawn as dashed and dotted lines. Node 7 represents the call statement to the `sum` function. PICO uses yellow to highlight calling and called nodes. A *call* edge connects node 7 and 0 whereas node 1

is connected with a *return* edge to node 7. In addition, the results of the preprocessing can be observed in node 7, variable `a` was replaced with a constant 3.

```
int sum(int a, int b)
{
  return a+b;
}

int main()
{
  int a = 3;
  int result = 0;

  for (int i = 0; i < 10; i++)
  {
    result += sum( a, i );
  }

  return result;
}
```

Listing 3.9: Examplary code to demonstrate PDG construction.

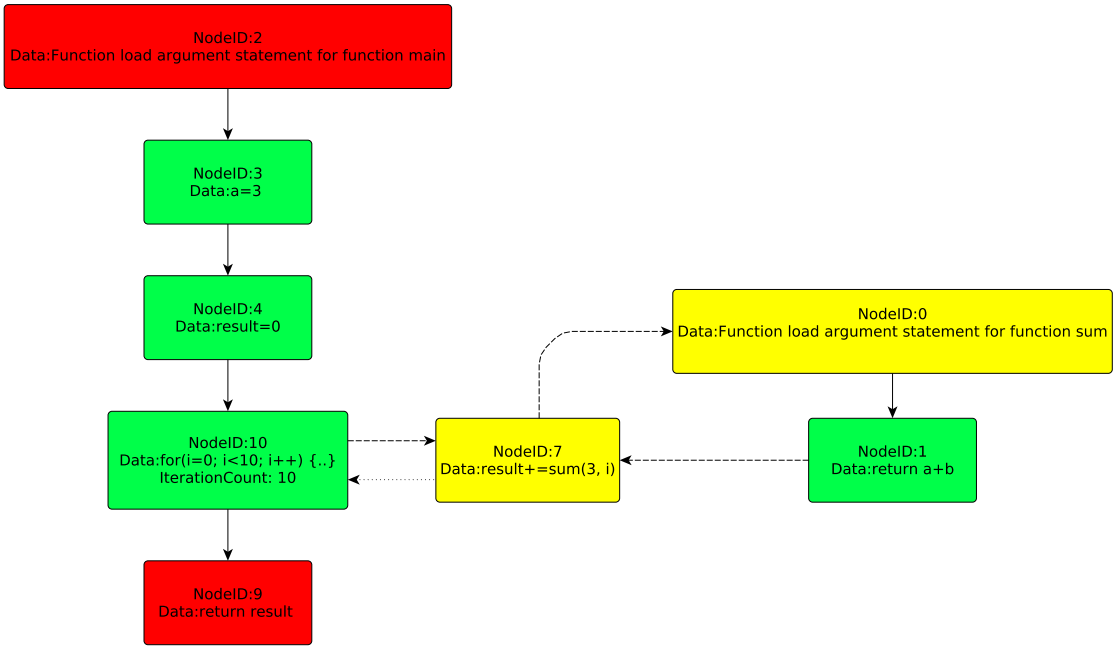


Figure 3.14: Interprocedural Control Flow Graph constructed by PICO for Listing 3.9.

After the CFG is constructed, PICO analyzes data dependencies. The ICD-C provides static data flow analysis, and here, PICO uses its *liveness*, *def-use*, *alias* and *points-to* analysis modules. Especially the employed alias and points-to analysis developed in collaboration with Daniel Cordes and Florian Schmoll enable PICO to track data dependencies of pointers. The FEHLER project [FEH18] also utilizes this data flow

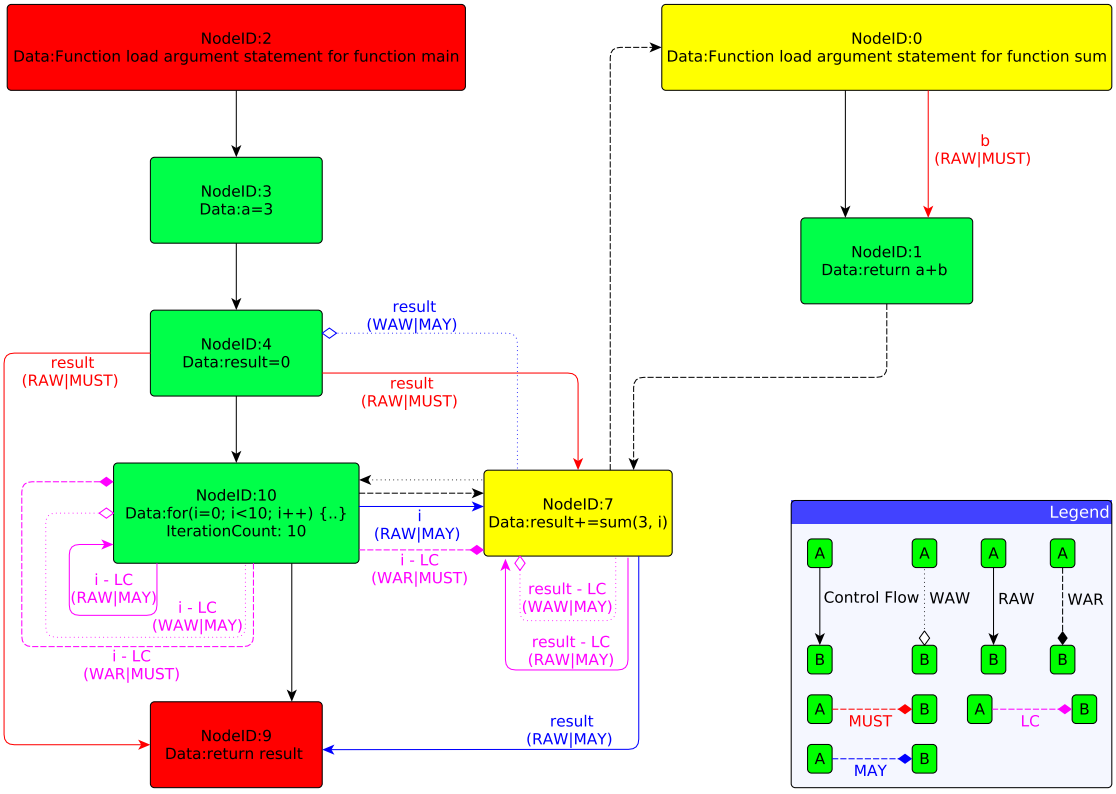


Figure 3.15: Program Dependence Graph constructed by PICO for Listing 3.9.

analysis. At this point PICO only conducts analysis on a symbolic level, value-based analysis is not implemented yet. This leads to conservative results especially for the array (memory) access analysis. In this case, PICO assumes that the complete array is manipulated or read.

Since interprocedural data flow analysis in combination with alias and points-to analysis is complex, time-consuming and tends to very conservative results, the user can select a simplified analysis, relying on the features of Parallelizable C (cf. Subsection 3.2.3). Then, PICO assumes that only symbols used as reference arguments are modified in the called function. In all benchmarks used in this thesis, this assumption was safe and produced valid dataflow analysis results. In addition to ICD-C's static dataflow analysis, PICO performs a deep loop analysis to detect loop-carried dependencies using an iterative work-list algorithm. A good loop-carried dependency detection is crucial to parallelize loops safely. With the results obtained by the static data flow analysis, PICO enhances the CFG.

Figure 3.15 shows the resulting PDG. The color of the edges denotes if a data dependency results from the alias analysis or the loop-carried dependency analysis. A *MAY* dependency represents aliasing of pointers that may occur during execution and a *MUST* denotes aliasing that must occur during execution and information provided by *MUST* edges dominate *MAY* edges. Different arrowhead and line styles are used to distinguish between the types of dependency. A white diamond arrowhead with a dotted line represents a WAW dependency. A solid line and arrow arrowhead shows a RAW

dependency whereas WAR dependencies are drawn as dashed lines with a filled diamond arrowhead. In Figure 3.15 node 10 has loop-carried dependencies to itself through the symbol `i`. This stems from the fact that `i++` is executed at the end of the loop and thus creates a loop-carried dependency to the guard `i < 10` statement. Node 7 passes `i` by value to the `sum` function thus no data dependency edges exist. In the case that `i` is passed by reference, PICO would add necessary data edges. In this example, the analysis detected that the argument `a` is constant inside the `sum` function and thus does not produce a data dependency between the function load argument statement and the return node.

3.6.1.2 Parallel Region Extraction

After the PDG is constructed, PICO scans all statements for annotations (cf. Section 3.5) starting with the innermost statements in the hierarchy. This bottom-up approach enables nested parallelism. Once PICO encounters such annotation, it extracts the type of parallelism and the assigned processor mapping. In case of data-level or hybrid pipeline parallelism, it also extracts the allocation of iterations. If no mapping is specified, PICO uses a heuristic taking the characteristics of the target platform into account. In the homogeneous case, PICO uses round-robin scheduling for tasks and iterations and therefore, PICO partitions the iteration space equally. If the iteration space cannot be equally divided by the number of available processors, one partition might be smaller as the others. In case of heterogeneous systems, PICO assigns a parallel region to a group of similar processors, like same Instruction Set Architecture (ISA) processors. The MACCv2 framework allows a performance comparison of processors.

With this information, PICO partitions the iteration space and maps larger workloads to more capable processors. With the extracted allocation knowledge, PICO traverses the PDG and assigns tasks to the nodes. For faster access, a separate data structure stores task to processor and iteration to task mappings. In addition, PICO creates a table of all parallel regions and their relation, like nested regions or which regions may run in parallel.

The main loop from Listing 3.9 can be parallelized and Listing 3.10 shows the necessary annotations to instruct PICO to use data-level parallelism. It is obvious that this parallelization requires synchronization between the parallel tasks. Figure 3.16 shows the updated PDG with assigned tasks. Data dependence edges are omitted in this graph for a better visualization. The nodes representing the loop are assigned to task 1 and 2 and the other nodes to the main task 0. Since PICO did not find nested parallelism in the called function, these nodes are not particularly assigned to a task. The calling relation determines the mapping to a task and this information is sufficient for the actual parallelization process. The implementation phase takes care that the function code is only included in the source code for tasks calling this function.

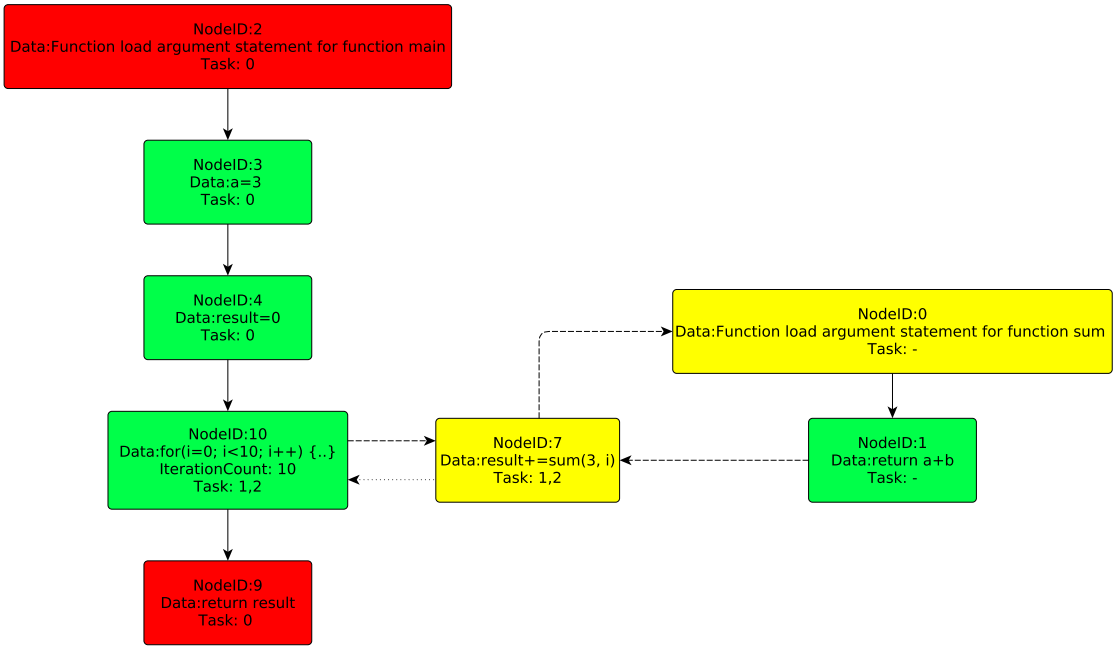


Figure 3.16: Program Dependence Graph with task mapping.

```
#pragma pico parallel for chunks=5,3 interleaved
for (int i = 0; i < 10; i++)
{
  result += sum( a, i );
}
```

Listing 3.10: Examplary code with PICO annotations.

3.6.1.3 Task Graph Construction

Finally, after PICO constructed the PDG and identified all parallel regions, PICO can construct the parallel task graph representing the technical realization of the application model (cf. Section 3.2). Algorithm 3.2 provides a brief algorithmic description on the task graph construction. To construct the task graph PICO traverses all parallel regions and their nodes from the innermost nested region upwards. Around each parallel region, PICO inserts *fork* and *join* nodes. These nodes split and join the sequential control flow into a parallel flow. Each task starts with a *taskIn* node and exits with a *taskOut* node. These nodes bundle the control flow entering and leaving a task. Data flowing from or to the parent task are routed through these nodes. In case that several sections belong to one task, PICO connects these sections with proper edges to construct a connected task. PICO duplicates nodes assigned to more than one task and adjusts the successor and predecessor nodes and data flow accordingly. Data flow crossing task boundaries indicates that a synchronization is necessary.

If the parallel region splits a loop and contains data dependencies crossing task boundaries, PICO performs an additional data flow analysis. Therefore, to implement

Algorithm 3.2 Task Graph Construction

```
procedure constructTaskGraph(PDG pdg, ParallelRegions parRegions)
    taskGraph ← pdg
    sortedRegions ← sortBottomUp(parRegions)

    for all region ∈ sortedRegions do
        root ← region.getRootNode()
        taskGraph.insertForkJoin(root)

        // add nodes for tasks
        for all task ∈ region do
            firstNode ← task.getFirstNode()
            lastNodes ← task.getLastNodes()
            taskGraph.insertTaskIn(firstNode)
            taskGraph.insertTaskOut(lastNodes)
            taskGraph.connectSections(task)

            if region.isPipeline() ∨ region.isParallelLoop() then
                taskGraph.insertLoopHead(task, region)
            if region.duplicate() then
                for all node ∈ pdg.getNodesAssignedToTask(task) do
                    taskGraph.duplicateNodes(node, task)
                end for
        end for
    end for

    // Insert communication
    for all ∀e ∈ dataEdges : sourceTask(e) ≠ destinationTask(e) do
        t_s ← sourceTask(e)
        t_d ← destinationTask(e)
        if t_s.isParentOf(t_d) then
            taskGraph.addDataToTaskIn(e, t_d)
        else if t_d.isParentOf(t_s) then
            taskGraph.addDataToTaskOut(e, t_s)
        else
            taskGraph.insertCommunicationNode(e)
    end for
    return taskGraph
```

the communication between parallel tasks, PICO calculates the dependent iterations for each symbol. PICO uses virtual loop unrolling to calculate the set of dependent iterations for each data dependency crossing task boundaries. A dependent iteration is defined as follows:

Definition 3.10 (Dependent iteration):
Let l be a loop, I the iteration space of l and t_i a task. Iterations $i, j \in I$ are dependent in symbol a if i and j have a data dependency over a and i is mapped to t_i and j to t_j with $t_i \neq t_j$.

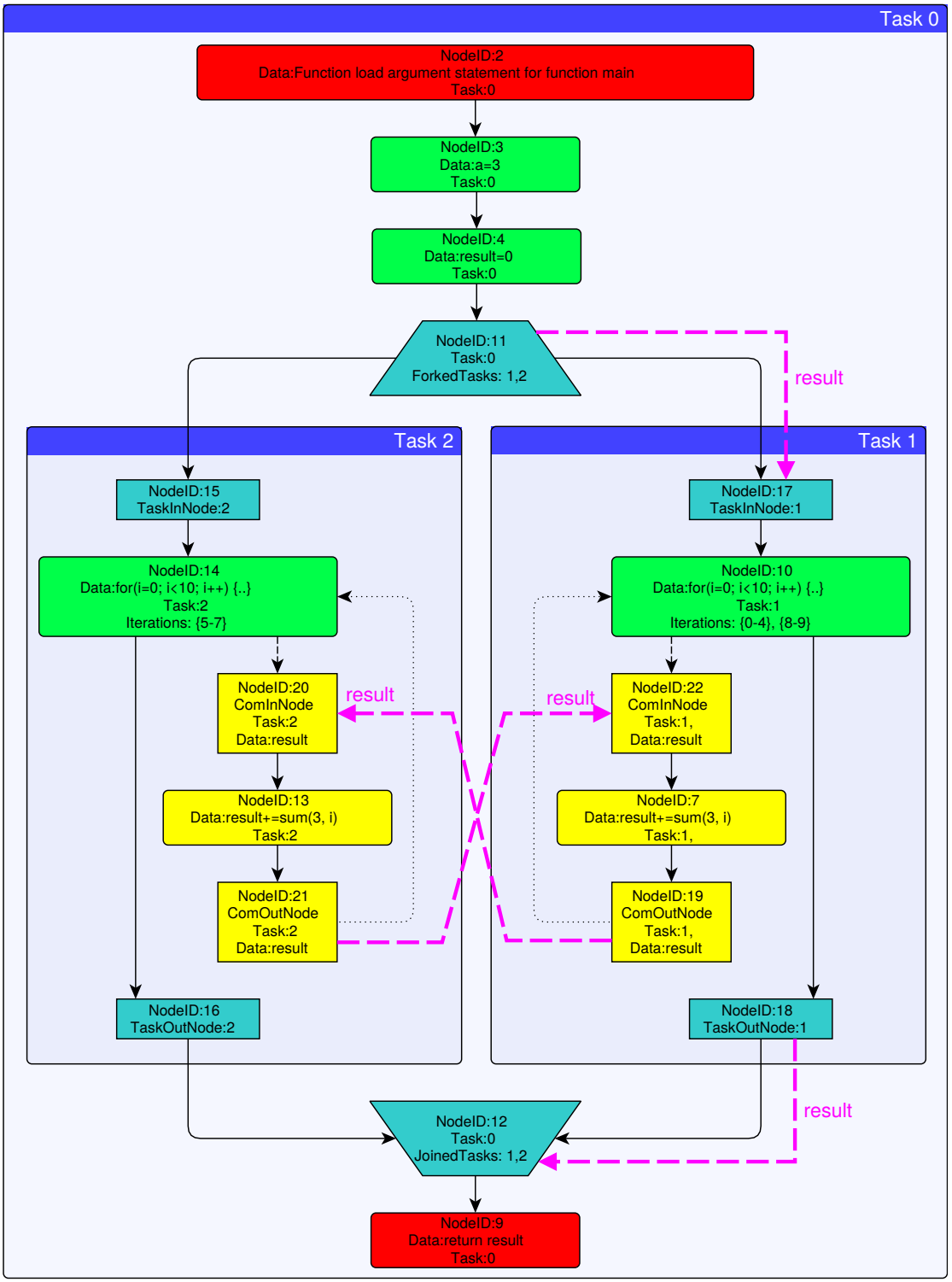


Figure 3.17: Task graph with highlighted data flow of symbol `result`.

With this knowledge, PICO inserts communication nodes directly after the source and before the target node of the data dependency edge. Using this strict synchronization, the parallelism implemented by PICO is correct-by-construction. Each communication node stores the set of dependent iterations, the symbol as well as the receiving/sending node (cf. Subsection 3.2.2).

Figure 3.17 shows the task graph of the example application from Listing 3.10. For a compelling visualization, the graph does not contain the nodes of the sum function and the true data edges. *Fork* and *join* nodes are drawn as trapezoid, *taskIn* and *taskOut* as cyan boxes with shared edges and *comOut* and *comIn* nodes as yellow boxes. We artificially added purple dashed lines to visualize the important data flow of the `result` symbol. As depicted, `result` flows from task 0 to task 1 through the *taskIn* node. Due to the partitioning of the iteration space, task 1 passes `result` to task 2 in iteration four. Task 2 sends `result` in iteration seven back to task 1. `Result` flows back through the *taskOut* node of the parent task 0. At this stage, PICO only operates on graphs and implementation details are irrelevant. During the implementation phase, PICO specifies the type of communication in more detail.

3.6.2 Implementation Phase

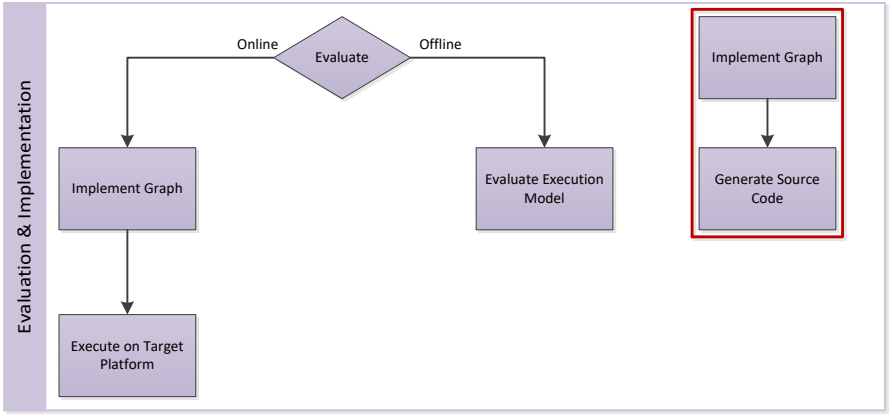


Figure 3.18: Implementation phase.

The implementation phase transforms the task graph back to C code. As Figure 3.18 depicts, this step is two-folded. First, the phase configures the graph, generates all statements internally, configures communication channels and integrates everything into a parallel IR. Then, ICD-C can generate the actual C code.

PICO starts with a list of tasks and generates an implementation for each task. Algorithm 3.3 provides a comprehensive representation of the following textual description regarding the task function generation. Basically, a task represents a subtree of the task graph and thus PICO can implement them independently. For each task, PICO generates a C function. PICO uses different implementation strategies depending on the type of parallelism. For example, if a loop is parallelized, PICO generates a new loop statement and constructs the loop head according to the iteration space partitioning. PICO introduces *execution guards* if the iteration space for a task is irregular in absence of a common iteration step size. Listing 3.10 shows an example using such irregular iteration space. PICO then processes the subtree and generates IR statements for all nodes. Nodes, originally from the sequential application, can be simply duplicated and assigned to the task function or loop respectively. It might be necessary to create new

Algorithm 3.3 Task Function Implementation

```
procedure implementGraph(IR ir)
    for all task ∈ Tasks do
        func ← ir.generateEntryFunction(task)
        if task.isPipeline() ∨ task.isParallelLoop() then
            adjustIterationSpace(task.getRootLoop())
        for all n ∈ task.getNodes() do
            if node.isCommunicationNode() then
                if node.dependentIteration ≠ task.dependentIterations then
                    func.insertGuard(node.dependentIteration)
            func.insert(node.generateC())
        end for
    end for
```

local copies of the symbols used in the statements. Besides the proper placement of statements, PICO also takes care that all (new) symbols used in the statements are assigned to the associated symbol tables.

For communication nodes, PICO inserts proper `read_channel` and `write_channel` API calls to the runtime library. In the PA4RES methodology, we assume that channels are used exclusively for one symbol. In this chapter, communication optimization is disabled, thus PICO configures the channel with predefined settings for the capacity and channel types. Chapter 4 presents PICO's communication optimization capabilities. If data synchronization between different loop iterations or pipeline stages is required, communication API calls must only be executed at dependent iterations. PICO achieves this by adding *iteration guards* surrounding these calls. In the default configuration, PICO implements data exchange between parent and child tasks using global `struct` variables. However, this can also be changed to a channel-based implementation by the user or PICO's communication optimization.

Task creation and management is outsourced to the lightweight runtime. Thus, PICO inserts `fork` and `join` API calls for corresponding nodes. A runtime library might generate a task dynamically or execute a previously generated task. The simulator-based target platform uses the latter approach. PICO instructs MACCv2 with internal annotations to generate proper linker scripts and target compiler configurations for the parallel tasks. After PICO generated the parallelized IR, the PA4RES framework emits actual C files.

Listing 3.11 shows the resulting C source code for task 1 derived from Listing 3.10. In this case, PICO used a global `struct __toTask1` to synchronize data from the parent task 0 to its child task 1. For systems without a shared memory, channels could be used. In this example, the iteration space is portioned irregularly thus an *execution guard* is required. In the case that the loops were partitioned equally, the loop head would increase `i` by the partition size. The *execution guard* used in this example ensures that task 1 only executes iterations specified in `chunks` clauses. As Figure 3.17 reveals, data must be synchronized between the parallel running tasks. Additional *guards* are

necessary to ensure a correct synchronization. Data needs to be communicated from task 1 to task 2 in the 5th and read back in the 9th iteration. Finally, task 1 returns data to the parent task using an additional global data structure.

```
void task_1() {
  int result=__toTask1.result;
  for (int i=0; i<10; i++) {
    if (i>=0u && i<=4u || i>=8u && i<=9u) {
      if (i == 8u) {
        pico_read_channel(2u, (&result));
      }
      result+=sum(3, i);
      if (i == 4u) {
        pico_write_channel(1u, (&result));
      }
    }
  }
  __fromTask1.result=result;
}
```

Listing 3.11: Implementation of task 1 with PICO API calls.

Section A.3 presents a detailed description of the runtime library and API. In the following, limitations and possible solutions of the presented approach are discussed.

3.6.3 Limitations

As previously discussed, PICO utilizes several graph structures during the parallelization process. Those graphs grow with the number of statements in the original source code and the number of parallel tasks. It is obvious that these graphs can explode and exceed the available resources. During the implementation of PICO, we thus considered this and carefully selected very light weight implementations of the graphs. PICO handled the graphs for the benchmarks analyzed in this thesis without issues. However, PICO provides a workaround to deal with large applications. Usually PICO processes all C files of the application even if only one file contains parallelism. To reduce the growth of the graph, PICO provides methods to only process files with parallel regions and considers all other files as black box libraries. The PA4RES framework already supports object files which are linked at the final step to the parallelized application. This approach has the drawback that the data flow analyses do not know what happens inside the called functions. As previously discussed (cf. Subsection 3.6.1.1), PICO then assumes that symbols passed as references are modified inside the function. This might lead to pessimistic analysis results. In such a case, new annotations could be used to integrate user knowledge into the analysis process. Finally, PICO relies on safe and precise data analysis and might mispredict data dependencies in absence of good analysis results.

Benchmark	Description
Adpcm	Adaptive Differential Pulse Code Modulation
Boundary-Value-Problem	Differential equation with boundary conditions
Compress	Discrete cosine transformation for image compression
Edge Detect	Edge detection on gray scale images
Filterbank	Pipeline of filter stages, including convolution, down and up sampling
FIR-Filter	Finite Impulse Response filter
Lattice-Filter	Digital normalized lattice filter
Matrix Mult.	Matrix multiplication
Spectral	Calculates the power spectral estimate of speech using periodogram averaging
JPEG	JPEG encoder
PAMONO Preprocessing	Preprocessing of the PAMONO virus detection software

Table 3.1: Benchmark description - benchmarks taken from UTDSP benchmark suite [Lee08], SNU real-time benchmark suite [SNU17] and preprocessing of the PAMONO virus detection software [Nol14].

3.7 Evaluation

This section evaluates PICO in three different aspects. First, we present how PICO is tested from the implementation side to ensure it creates valid parallel applications. Then, we will present an evaluation regarding the usability of our proposed annotations. Finally, we evaluate the performance of the parallelized application. Besides synthetic tests, we used applications from the UTDSP benchmark suite [Lee08] and SNU real-time benchmark suite [SNU17] containing representative real-world embedded applications and algorithms. In addition, the preprocessing phase of the PAMONO virus detection software is used [Nol14]. The preprocessing contains several image processing steps like background elimination or noise reduction and was specifically designed for the PA4RES framework. Table 3.1 summarizes all used benchmarks.

3.7.1 Proof of Concept and Implementation

As previously described, PICO internally performs several graph transformations to parallelize a sequential application. These graph transformations satisfy the data dependencies by construction since communication is added after modifying and before accessing data, following the application model defined in Section 3.2. However, to validate the implementation of these transformations, PICO undergoes dozens of synthetic and real world tests. Synthetic test cases are manually written applications aiming to provoke flaws in PICO's implementation. Parallelization of synthetic tests usually does not gain a significant speedup, e.g. due to intense synchronization efforts. For all test cases, PICO parallelizes the applications and executes them on the target platform. The test applications produce some kind of result, e.g. debug output between calculations with intermediate results. These results are then compared with the results from a sequential (unmodified) execution of the same application. Jenkins [Jen17], a continuous integration

server, automatically performs these tests every time code changes are committed to the code repository.

```
int a[10] = {0,0,0,0,0,0,0,0,0,0};
int b = 0, adder = 0;
#pragma pico parallel pipeline for num_threads(4)
for (int i = 0; i < 10; i++)
{
  //first inner loop
  #pragma pico section taskid=1 chunks=1,1 interleaved
      ↪ processor={0},{1}
  for (int j = 0; j < 10; j++)
  {
    a[j] = j + adder;
    adder++;
  }
  //second inner loop
  #pragma pico section taskid=2 chunks=1,1 interleaved
      ↪ processor={2},{3}
  for (int k = 0; k < 10; k++)
  {
    b = b + a[k] + adder;
    adder++;
  }
}
```

Listing 3.12: Synthetic hybrid pipeline benchmark with complex data dependencies.

Listing 3.12 shows one sophisticated synthetic hybrid pipeline test case. Here, the test specifies two pipeline stages which are then duplicated provoking complex data dependencies. PICO needs to detect which data dependencies must be satisfied in which iteration/pipeline stage as well as which tasks exchange data with the parent task. Figure 3.19 illustrates the complex data flow for the variable `adder`. As these tests show, PICO is able to detect and implement complex data dependencies and parallel structures.

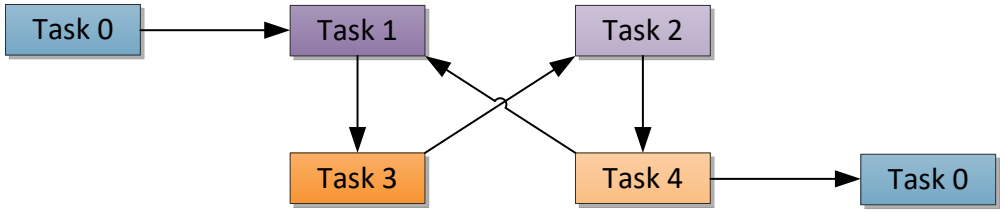


Figure 3.19: Data flow for `adder` in parallelized synthetic hybrid pipeline benchmark. Task 1 and 2 originate from the first section and Task 3 and 4 accordingly from the second section.

3.7.2 Usability Analysis

PICO processes either source code with annotations generated by PAXES or user added directives. In the latter case, a good usability of PICO is important and was the driving

Solution	Run time (s)	Energy (mJ)	Speedup	Energy factor
Sequential	4.674	175.115	1	1
Manual	2.337	87.558	2	0.5
Parallel for	3.002	110.213	1.56	0.63
Hybrid pipeline	2.945	117.817	1.59	0.67

Table 3.2: PICO-based compared to manual and time-consuming parallelization for PAMONO preprocessing executed on the homogeneous simulation-based platform.

factor in the development of the application model and directives. In Section 3.5 we already highlighted the simplicity of the PICO approach especially for hybrid pipeline parallelism. In Subsection 3.5.3 we discussed that loop-carried dependencies and the correct synchronization are crucial factors for a parallelization of the `Spectral` analysis benchmark. With PICO, only a few directives are necessary to extract such complex parallelism. The next subsection presents the performance analysis for PICO and with it provides additional examples using directives.

```
#pragma pico parallel pipeline for num_threads(4)
for (i = 0; i < 8; i++) {
  float Vect_H[256]; //output of the H
  float Vect_Dn[32]; //output of the down sampler
  float Vect_Up[256];// output of the up sampler
  float Vect_F[256];
  #pragma pico section chunks=1,1,1 interleaved
  {
    //convolving H
    ...
    //Down Sampling
    ...
    //Up Sampling
    ...
    //convolving F
    for (j = 0; j < 256; j++) {
      Vect_F[j] = 0;
      for (k = 0; ((k < 32) & ((j - k) >= 0)); k++) {
        Vect_F[j] += F[i][k] * Vect_Up[j - k];
      }
    }
  }
  //adding the results to the y matrix
  #pragma pico section
  for (j = 0; j < 256; j++)
  {
    y[j] += Vect_F[j];
  }
}
```

Listing 3.13: Annotated Filterbank benchmark.

Listing 3.13 shows an outline of the instrumented `Filterbank` benchmark. For simplicity, Listing 3.13 omits most of the filter implementations. This application applies a set of filters to an input vector and stores the result in the output vector `y` and benefits from pipeline parallelism. Due to extracted data dependencies, PICO will add communication of `Vect_F` between the first stage and the second stage. In addition, the first pipeline stage is duplicated so that three instances share the workload resulting in four parallel tasks. This example highlights that developers are able to parallelize complex algorithms successfully with PICO.

Stefan Noll ported the PAMONO virus detection preprocessing pipeline to the simulation-based platform during his bachelor thesis [Nol14], supervised by the author of this thesis. In his thesis, he analyzed different parallelization strategies manually in a time-consuming process. Already the safe implementation of a parallel version took weeks. However, to demonstrate PICO's capabilities we successfully parallelized the preprocessing algorithm with only four annotations. We evaluated loop- and hybrid pipeline-level parallelism. The parallel solutions were derived within a working day whereas the majority of the time was spent to understand the preprocessing algorithm implementation. Table 3.2 shows the results of this experiment. The manual implementation performs best and achieved a speedup of 2. However, PICO was able to generate a remarkable speedup of almost 1.6. PICO's results are not as good as the manual tuned implementation but were derived much faster, within hours in contrast to weeks.

Listing 3.14 lists a comparison of necessary PICO and OpenMP directives to parallelize a loop from the `Matrix Multiplication` benchmark. The developer is in charge of detecting data dependencies and taking care of correct handling in OpenMP's relaxed synchronization model. The advantage of PICO is that data dependencies are detected automatically and thus, PICO is perfectly suited for rapid prototyping and incremental parallelization.

```
#pragma pico parallel for num_threads(4)
// versus
#pragma omp parallel for num_threads(4) private(sum,i,j,k)
    ↪ firstprivate(a_matrix, b_matrix)
for (int i = 0; i < A_ROW; i++) ...
```

Listing 3.14: Comparison between PICO and OpenMP clauses.

3.7.3 Performance Analysis

To analyze the performance of the code generated by PICO, we used PAXES to detect and annotate parallelism in the benchmarks. This results in a set of Pareto-optimal solutions (cf. Definition 1.1) for each benchmark (cf. Subsection 2.3.2). In this case, speedup and energy consumption of the benchmarks can be conflicting objectives. These trade-offs are very important in the design of low-power embedded systems and software. In the end, the user decides which solution fits best to the scenario. We selected one solution for each benchmark randomly as input for PICO.

The simulation-based platforms (cf. Figure 2.2) act as the main target architecture in this evaluation. For the homogeneous platform, all cores are set to a frequency of 500 MHz. In the heterogeneous case, the first two processors are set at 500 MHz, the third is running at 250 MHz and the last with 100 MHz. A high-level cost model (cf. Appendix A.1) was used to calculate the energy consumption PAXES mostly chooses data-level parallelism but due to loop-carried dependencies for three benchmarks selected pipeline parallelism. For all experiments, we disabled the communication optimization part. Therefore, PICO implements all data synchronizations with a default configuration.

3.7.3.1 Homogeneous Experiments

Table 3.3 and Table 3.4 summarize the results of the parallelization process for the homogeneous system. The results achieved by PICO are en-par with previous results from the MNEMEE project as reported in [Cor13] using a proprietary solution. Since PICO detects data dependencies automatically, no manual data dependency management was necessary. The `Matrix Multiplication` benchmark is a good example that demonstrates that PICO is able to achieve a super-linear speedup. Unfortunately, the other benchmarks are not always parallelizable well. This is not an issue of PICO. It is more an algorithmic problem, e.g. in `Adpcm` only one small loop could be found to be parallelized leading to a small speedup of just 1.12. However, even small improvements might be a win in some circumstances. PICO achieved noticeable speedups for the benchmarks requiring communication.

To get an better understanding of the results, we conducted another experiment using OpenMP to parallelize the benchmarks using the Odroid-XU3 as the target platform. We followed the same parallelization strategies as in the previous experiment. Since PICO reported the data dependencies, we were able to model the dependencies in OpenMP accordingly. Hence, we added necessary *private* and *firstprivate* clauses. For benchmarks where PICO added communication channels, OpenMP `dependent task` constructs are used. Compiler optimizations were disabled and the number of threads was set to four. We executed each benchmark 1000 times, measured run times and energy consumptions were averaged. We used *perf* to measure the run time and the EnergyMeter (cf. Subsection A.2.1) to measure the energy consumption. We analyzed two setups, one without any scheduling restrictions (cf. Table 3.5 and Table 3.6) and one where the tasks are restricted to the Cortex-A7 (cf. Table 3.7 and Table 3.8). Without restrictions, the operating system decides where to execute the threads, usually depending on the current load. This can be beneficial if execution gets delayed, e.g. due to dependencies. Then, this task can be scheduled to the energy efficient cores. Limiting the threads to the Cortex-A7 should result in good energy efficiency but maybe longer execution times.

As the results indicate, the benchmarks do not perform as well as in the previous experiments. On one hand, the Odroid-XU3 platform is far more powerful than the simulation-based ones. On the other hand, the benchmarks are the same and thus the computational load. Thus, OpenMP's management overhead dominates the run time.

Benchmark	Com Channels	Sequential Run time (ms)	Sequential Energy (μJ)	Parallel Run time (ms)	Parallel Energy (μJ)
Adpcm	0	2.55	76.67	2.28	152.72
Boundary-Value-Problem	0	45.49	1242.66	13.75	632.59
Compress	0	120.17	4223.98	33.63	2151.74
Edge Detect	0	147.31	7582.11	105.68	6143.27
Filterbank	3	119.11	4595.60	36.52	1515.77
FIR-Filter	0	8.35	344.30	3.58	233.43
Lattice-Filter	0	3.07	121.12	2.00	144.62
Matrix Mult. 80x80	0	225.69	9709.25	56.35	2967.16
Spectral	6	6.90	214.52	5.84	425.46
JPEG	3	76.75	2696.46	27.95	1548.76

Table 3.3: Results for PICO-based parallelization executed on a homogeneous system with four ARM cores, simulated with Virtualizer. Identification of parallelizable loops done by PAXES.

Benchmark	Speedup	Energy Factor
Adpcm	1.12	1.99
Boundary-Value-Problem	3.31	0.51
Compress	3.57	0.51
Edge Detect	1.39	0.81
Filterbank	3.26	0.33
FIR-Filter	2.33	0.68
Lattice-Filter	1.54	1.19
Matrix Mult. 80x80	4.01	0.31
Spectral	1.18	1.98
JPEG	2.75	0.57

Table 3.4: Comprehensive speedups and energy factors for Table 3.3.

Therefore, we increased the computational workload of the `Matrix Multiplication` benchmark. We raised the size of the matrices from 80×80 up to 800×800. This changes the ratio of the OpenMP overhead and computational load and the resulting speedup is as expected.

The enlarged `Matrix Multiplication` benchmark demonstrates the two different performance characteristics of the two processor types used in the Odroid-XU3. The run time of the parallelized version on the Cortex-A15 is roughly 4s and almost 18s on the Cortex-A7. As expected, the fast core performs better in terms of run time, however, the energy consumption tells a different story. The Cortex-A7 just consumed 8J whereas the Cortex-A15 burned almost 18J. This emphasizes the idea of heterogeneous platforms to trade performance against energy consumption.

In an additional experiment, we ported PICO's runtime library to the Odroid-XU3 platform and integrated that system into the PA4RES framework. It took us less than one day to add this new platform to the framework. The integration of the poper MACCv2 classes and configurations to represent the Odroid-XU3 took the majority of

Benchmark	Sequential		Parallel	
	Run time (ms)	Energy (mJ)	Run time (ms)	Energy (mJ)
Adpcm	5.23	16.29	5.05	16.87
Boundary-Value-Problem	6.52	18.57	5.98	19.67
Compress	8.00	24.42	6.87	27.39
Edge Detect	10.01	29.85	8.62	37.80
FIR-Filter	3.6	11.41	4.87	15.55
Filterbank	10.41	33.00	9.94	31.14
Lattice-Filter	3.57	11.34	5.04	16.82
Matrix Multiplication 80x80	9.78	28.12	7.75	31.90
Matrix Multiplication 800x800	15,935.11	62,484.40	4,335.91	17,684.60
Spectral	4.08	13.05	5.29	16.88
JPEG	11.45	32.03	8.79	30.24

Table 3.5: Results for OpenMP-based parallelization executed on the Odroid-XU3 without scheduling restrictions.

Benchmark	Speedup	Energy Factor
Adpcm	1.03	1.03
Boundary-Value-Problem	1.09	1.06
Compress	1.16	1.12
Edge Detect	1.18	1.27
FIR-Filter	0.74	1.36
Filterbank	1.05	0.94
Lattice-Filter	0.71	1.48
Matrix Multiplication 80x80	1.26	1.13
Matrix Multiplication 800x800	3.68	0.28
Spectral	0.77	1.29
JPEG	1.30	0.94

Table 3.6: Comprehensive speedups and energy factors for Table 3.5.

the time. Section A.3 gives more details regarding the API and runtime. To further ease the evaluation process, we added cross compilation and remote execution with measurement capabilities to PA4RES. Hence, the framework transmits and executes the cross compiled binary via ssh to the Odroid-XU3. PICO collects the measurement results and integrates them, thus they can be used in later optimization phases. Since the used benchmark set is not ideal for the powerful Odroid-XU3, we conducted this experiment only with the large `Matrix Multiplication` benchmark. The result was almost identical to the one reported by the OpenMP experiment. Generally, the runtime is reduced and therefore the energy consumption. This experiment suggests that PICO achieves a comparable performance to OpenMP.

3.7.3.2 Heterogeneous Experiments

After we demonstrated PICO's capabilities on homogeneous systems, we shift the focus towards heterogeneity. In the following, we show how PICO meets the requirements of

Benchmark	Sequential		Parallel	
	Run time (ms)	Energy (mJ)	Run time (ms)	Energy (mJ)
Adpcm	9.61	6.58	9.59	7.50
Boundary-Value-Problem	12.14	8.22	10.79	9.17
Compress	21.38	13.98	13.70	11.59
Edge Detect	27.80	16.81	21.58	18.658
FIR-Filter	7.38	6.52	9.27	13.57
Filterbank	22.5	14.37	22.42	14.37
Lattice-Filter	7.38	6.19	9.34	7.28
Matrix Multiplication 80x80	26.69	16.04	15.33	12.38
Matrix Multiplication 800x800	34,158.54	8,861.61	17,951.20	8,032.08
Spectral	8.53	6.80	8.25	6.85
JPEG	18.99	12.42	18.83	12.52

Table 3.7: Same configuration as in Table 3.5 except, tasks are restricted to the Cortex-A7.

Benchmark	Speedup	Energy Factor
Adpcm	1.01	1.14
Boundary-Value-Problem	1.13	1.12
Compress	1.56	0.83
Edge Detect	1.29	1.11
FIR-Filter	0.79	2.08
Filterbank	1.01	0.99
Lattice-Filter	0.79	1.18
Matrix Multiplication 80x80	1.74	0.77
Matrix Multiplication 800x800	3.85	0.45
Spectral	1.03	1.01
JPEG	1.01	1.01

Table 3.8: Comprehensive speedups and energy factors for Table 3.7.

heterogeneous low-power embedded systems with three benchmarks in detail. Especially, we discuss the `Spectral`, `Filterbank` and `JPEG` benchmarks in detail. We selected these benchmarks as they represent complex parallelization with pipeline parallelism and communication. As in the previous experiments, the (Pareto-optimal) parallelization decisions are the result of the automatic parallelization process by PAXES. Besides the performance of the implementation, we compare PICO's static iteration mapping heuristic with PAXES precise mapping.

Listing 3.15 shows the main function of the application annotated with PICO directives. Only four PICO directives are necessary to describe such a complex pipeline parallelization. Subsection 3.5.4 and Subsection 3.5.3 also discuss this benchmark. The statements of the first loop are grouped into two disjoint pipeline stages. The first pipeline stage is divided into three concurrent tasks, assigned to processors 1 or 2, and 3 and 4. The second pipeline stage is mapped to one of the 500 MHz processors (1 or 2). The second parallel region uses the `parallel for` construct to distribute the iterations of the loop over three tasks and processors. In this case, the fourth processor

is not used and can be power-gated to reduce the energy consumption. This emphasizes another difference between parallelization for embedded systems compared to HPC with respect to multiple objectives. For embedded systems, it might be beneficial to not extract as much parallelism as possible in order to achieve additional, non-functional objectives such as the reduction of the overall energy consumption. In this case, the iteration scheduling generated by PICO's heuristic is identical to the precise mapping generated by PAXES. We added the iteration mapping clauses (chunks) to Listing 3.15 for visualization purposes. Hence, these clauses are not included in the file passed to PICO.

```
#pragma pico parallel pipeline for num_threads(4)
for (int i = 0; i < 16; ++i) {
  float s_real[64]; float s_imag[64];
  #pragma pico section taskid=1 processor={1,2},{3},{4}
     ↪ chunks=5,3,1 interleaved
  {
    int index = i*4;
    for (int j = 0; j < 64; ++j) {
      s_real[j] = input_signal[(index+j)] * hamming[j];
      s_imag[j] = zero;
    }
    fft(s_real, s_imag);
  }
  #pragma pico section taskid=2 processor={1,2}
  {
    for (int j = 0;  j < 64; ++j) {
      mag[j] = mag[j] + (s_real[j] * s_real[j]
                         + s_imag[j] * s_imag[j]) / 4096;
    }
  }
}
#pragma pico parallel for num_threads(3) processor={1,2},{1,2},{3}
   ↪ chunks=26,26,12
for (int i = 0; i < 64; i++) {
  mag[i] = 10 * log10((mag[i] / 16 > 1.0e-14f ?
                       mag[i] / 16 : 1.0e-14f));
}
```

Listing 3.15: Parallelized main function of the Spectral Analysis benchmark for a embedded heterogeneous 4 core system. Iteration mapping was done by PICO's schedule heuristic and chunks clauses are just added to show the schedule.

The `JPEG` benchmark encodes images into jpeg files. Listing 3.16 shows the main computation loop of the parallelized source code. PAXES sliced the main loop into a pipeline with two stages and duplicated the first pipeline stage into 3 parallel tasks. The parallelizer did not split the second pipeline stage into parallel tasks, because only one processor is left and PA4RES consider systems without runtime scheduling per default.

Further, the listing shows PAXES' precise iteration mapping for the heterogeneous platform.

```
#pragma pico parallel pipeline for num_threads(4)
for(blk_y = 0; blk_y < RUN_BLK_Y; blk_y++) {
  for(blk_x = 0; blk_x < JPG_WIDTH/8; blk_x++) {
    UINT8   dctin [64];
    INT16   dctout[64];
    INT16   qzout [64];
    INT16   zzout [64];
    #pragma pico section taskid=1 iterations
        ↪ ={0,3,5,7,9,...,15,18,20,22,24,...,29},
        ↪ {1,4,6,8,16,19,21,23},{2,17}
        ↪ processor={1,2},{3},{4}
    {
      get_block(image + blk_x * 8 + blk_y * 8 * JPG_WIDTH, dctin);
      dct(dctin, dctout);
      quantize(dctout, qzout);
    }
    #pragma pico section taskid=2 processor={1,2}
    {
      zigzag(qzout, zzout);
      jpgsize = huffman(zzout, jpgsize);
    }
  }
}
```

Listing 3.16: Parallelized main function of the JPEG benchmark for an embedded heterogeneous 4 core system. Precise iteration mapping is generated by PAXES.

PAXES modified the `Filterbank` benchmark, presented in the usability analysis, slightly to take the structure of the target platform into account. The overall pipeline structure (cf. Listing 3.13) is identical but the first stage is now assigned to one of the two fast cores and the second stage on the other cores. This introduces additional data dependencies leading to 8 communication channels.

To examine a suboptimal parallelization for a heterogeneous platform, we executed the homogeneous version of the benchmarks on the heterogeneous platform (cf. Table 3.9 and Table 3.10, *hom2het*). Here, it is obvious that suboptimal load balancing results in reduced performance. To take the heterogeneity into account, we conducted experiments with PICO's automatic iteration scheduling *heuristic* and PAXES' *precise* iteration mapping. Overall, the run time could be reduced drastically with static load balancing. As expected, the energy consumption benefits as well from load balancing. For instance, the speedup of the `Filterbank` benchmark increased from 0.68 to 2.37 even with more communication channels. In addition, for the conducted experiments, the heuristic achieves almost identical speedups as the precise iteration mapping provided by PAXES.

The experiments shows that the slowdowns of `Spectral Analysis` and `Filterbank` in case of the *hom2het* experiments are similar but the energy consumption differs

Benchmark	Com Channels	Sequential Run time (ms)	Sequential Energy (μJ)	Parallel Run time (ms)	Parallel Energy (μJ)
Spectral hom2het	6	6.79	0.19	13.16	0.75
Spectral heuristic/precise	6	6.79	0.19	6.68	0.57
JPEG hom2het	3	70.76	3.63	73.91	4.53
JPEG heuristic	3	70.76	3.63	46.23	3.72
JPEG precise	3	70.76	3.63	38.86	3.46
Filterbank hom2het	3	108.57	6.10	160.13	4.18
Filterbank heuristic	8	108.57	6.10	46.05	1.81
Filterbank precise	8	108.57	6.10	45.76	1.78

Table 3.9: Results for heterogeneous experiments with different iteration mapping techniques.

Benchmark	Speedup	Energy Factor
Spectral hom2het	0.52	4.01
Spectral heuristic/precise	1.07	3.03
JPEG hom2het	0.96	1.25
JPEG heuristic	1.53	1.02
JPEG precise	1.82	0.95
Filterbank hom2het	0.68	0.69
Filterbank heuristic	2.36	0.30
Filterbank precise	2.37	0.29

Table 3.10: Comprehensive speedups and energy factors for Table 3.9.

drastically. Deeper investigations revealed that the different memory utilizations cause this gap. In case of the `Spectral` analysis, relatively large data is communicated through the memory whereas `Filterbank` is dominated by calculation and thus benefits from parallelization. This offers optimization opportunities exploited by PICO as presented in Chapter 4.

3.8 Conclusion

This chapter presented the overall PICO framework, our approach to tackle the tough challenge to parallelize sequential applications targeting modern multiprocessor systems with a strong focus on limited resources typical for embedded systems. The design philosophy allows developers to parallelize their applications with a high-level method. Embedded in the PA4RES methodology, PICO provides two ways to parallelize legacy sequential programs. In combination with PAXES, applications can be parallelized in a (semi) automatic tool flow without manual interaction. The other way provides a manual annotation-based approach. In this case, the design philosophy of PICO focuses on simplicity. With straightforward directives, inspired by the de-facto standard for shared memory parallelization, developers can parallelize their applications without deep knowledge of data dependencies and control flow.

PICO is integrated into the PAXES framework and thus uses MACCv2's target platform knowledge and the ICD-C compiler framework for the source-to-source transformations. Internally, PICO performs all operations on graph structures and thus is able to derive data dependencies between statements to ensure a safe parallel execution. In combination with annotations, PICO generates parallel source code including necessary data synchronization using a platform-independent lightweight runtime library calls. The evaluation demonstrated the benefits of PICO. Only a few directives were necessary to implement complex hybrid pipeline parallelism. The performance evaluation on the homogeneous system showed that PICO performs as expected. Using static load balancing, PICO successfully exploited the capabilities of the heterogeneous platform. The implemented iteration scheduling heuristic performs comparable to a precise iteration mapping generated by PAXES. This demonstrates PICO's strengths not only in combination with PAXES but also as an iterative development tool for rapid prototyping.

Chapter 4

PICO - Communication Optimization

Contents

4.1 Introduction

Parallelization is the key technique to improve the performance of applications in modern multi-core systems. Ideally, the application is partitioned in a way that all parts run independently. Unfortunately, data dependencies force these parts to synchronize which may delay the execution until all dependencies are met. This leads to a performance

degradation of the parallelized application. The previous chapter revealed that synchronization between parallel running tasks cannot be avoided. Especially pipeline parallelism indicates that data flows between concurrently running tasks. Despite this drawback, pipeline parallelism is sometimes the only option to leverage parallelism and improve the performance of a given application. Therefore, it is important to implement the data synchronization as efficiently as possible with respect to given resource limitations.

In the context of PA4RES, we identified important objectives such as energy consumption, run time and memory consumption. Following the PA4RES methodology, data synchronization between concurrently running tasks is realized with point-to-point FIFO-channels. In theory, these FIFO-channels have an infinite capacity and do not add additional delay. However, real systems have resource limitations like a limited memory capacity defining the maximum channel size or data transmission delay. In addition, different hardware and software implementations for channel-based communication vary in their properties and may introduce additional overhead. Further, the communication infrastructure available on the target platform exposes additional limitations, for instance bus contention. For systems with complex memory hierarchies, the channel to memory mapping influences the performance. A specific mapping may influence other channel parameters, for instance, using a small but fast memory for the communication limits the maximum capacity of the channel. In some cases a mapping may restrict the use of a specific channel implementation. On the software side, usually, multiple program variables must be synchronized across concurrent tasks. Therefore, the order of channel accesses as well as merging of multiple channels into one influence the performance of the parallelized application. Manual exploration of all these parameters in order to find an efficient solution is a complex, time-consuming and error prone process. PICO provides an easy to use method to explore the capabilities of the target platform and parallel application using an evolutionary algorithm. PICO creates a set of Pareto-optimal implementations of the channel-based synchronization where the user can select the best suiting solution. Thanks to the separation of task and channel implementation using abstract library calls, PICO can refine the communication independently from the task implementation.

This chapter is based on methods and results published in [NEM15b]. However, this thesis extends the evaluation and provides a more detailed description of the algorithmic and implementation details significantly. In addition, this chapter adds and evaluates a high-level execution model to speedup the exploration time. To summarize, this chapter provides a methodology to answer the following questions typically raised during the implementation of data synchronization following the PA4RES approach:

1. Which hardware or software implementation suits best?
2. Which FIFO-channel capacity prevents blocking?
3. Where to map a channel, e.g. which memory?
4. When to perform the communication in the execution flow?

5. Which channels should be combined?

6. How do different FIFO implementations affect the performance?

This chapter is structured as follows. Section 4.2 provides a brief overview of PICO's communication optimization approach. Section 4.3 discusses related work and compares it to PICO. Section 4.4 presents a detailed description of the internals of the optimization flow. Section 4.5 highlights the evaluation results and Section 4.6 summarizes this work and gives an outlook for future improvements.

4.2 PICO - Communication Optimization Approach

An efficient communication implementation is crucial for a parallel application. Thanks to the separation of task code and communication implementation, PICO can explore the target specific implementations independently. PICO's optimization algorithm tackles the mapping problem (cf. Definition 4.1) with a GA. The optimization phase works on the abstract application model (cf. Section 3.2). It is able explore various communication related parameters and automatically derives an optimized solution.

Definition 4.1 (Communication Mapping Problem):
Given a set of communication nodes $c \in C$, a set of channels $ch \in Ch$ and a set of communication infrastructure components $ci \in CI$, the communication mapping problem is to find an optimal mapping of $c \rightarrow ch \rightarrow ci$ such that the overall costs are minimal.

$$communicationCosts = \sum_{c \in C} (\sum_{ch \in Ch} costs(ch, c) + \sum_{ci \in CI} costs(ci, c))$$

$$overallCosts = executionCosts + communicationCosts$$

Costs can be execution time, energy consumption or memory space and depend on implementation details, e.g. channel capacity.

A GA belongs to the class of evolutionary algorithms and is inspired by the selection process observed in nature. They are simple to implement and able to explore complex solution spaces efficiently. The main components of a GA are the chromosome, the fitness evaluation and the genetic operations. A chromosome is basically an array of parameters, thus the genetic representation of the problem space. A individual is a specific solution and a set of individuals is a population. A fitness function evaluates how good an individual performs. The GA starts with an (random) initial population and applies several operations inspired by nature. A selection process chooses promising individuals from the population. These individuals are then merged in a cross-over process or mutated forming a new population. This iterative process continues until a certain termination criterion is met, for example a maximum number of populations, reaching a certain fitness level or an equilibrium. The main challenges of GA are an efficient encoding of the problem space into a gene representation, a good selection,

cross-over and mutation process to avoid local optimums and finally, a fast and accurate fitness evaluation.

Figure 4.1 shows the communication optimization flow using the PA4RES flexible tool flow (cf. Section 2.3). Starting with an annotated sequential application, either provided by PAXES (cf. Subsection 2.3.2) or manually instrumented by the user, PICO explores the solution space using a GA. The GA generates and proposes solution candidates to PICO for evaluation. PICO performs the fitness evaluation either on the target platform or using a high-level execution model. Finally, PICO returns a set of Pareto-optimal solutions to the user.

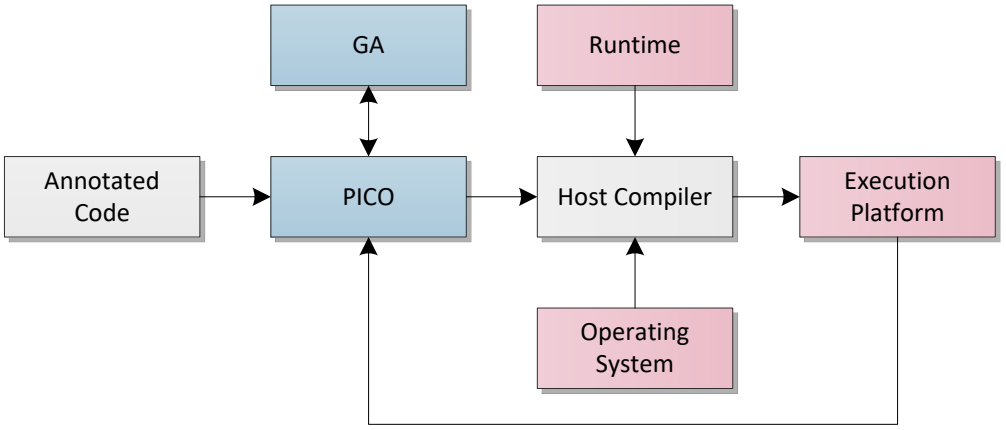


Figure 4.1: Communication optimization flow.

The algorithm is able to explore software FIFO and hardware implementations, if they follow the PICO communication API (cf. Section A.3). We created several prototype implementations for the channel-based synchronization which ship with PICO. Each implementation provides different advantages and disadvantages. We expect, that there is no FIFO implementation suiting all needs for any application. Therefore, we enabled the GA-based approach to explore the capabilities of these implementations with respect to multiple objectives. The set of predefined channels can be extended easily with new implementations satisfying PICO's API.

The GA can explore the memory hierarchies of a given system, for instance the SPM available in the simulation-based low-power embedded system (cf. Section 2.1). Beside these parameters, the GA also determines the capacity of each FIFO, and if multiple channels should be bundled to reduce management overhead. Thanks to the PA4RES framework and MACCv2, the optimization algorithm is aware of resource limitations like memory capacity. However, the user can restrict the memory usage artificially. This might be useful to store additional user-defined data into a fast but smaller memory or to explore different hardware configurations early in the system design exploration phase.

4.3 Related Work

Efficient data exchange between concurrently running tasks is tightly coupled to the pursued parallelization technique. Section 3.3 highlights the parallelization aspects of PICO related approaches. Most of these approaches assume fixed communication implementation or user interaction. However, if the already discussed related work provides additional insights into the realization of communication, we list them in this section as well.

FIFO channels are used in several models of computations, e.g. KPN, to synchronize data between concurrently running entities and to preserve a given execution order of the application. Therefore, mapping and implementation of FIFO channels onto different hardware platforms have been researched extensively. Nadezhkin et al. [NMS09] presented a specialized FIFO implementation to map KPNs onto the Cell BE platform. The heterogeneous Cell platform provides a PowerPC core and multiple synergistic processing elements connected though a bus. The system does not provide hardware FIFO facilities. Therefore, the global shared memory and private processor memory must be used for the synchronization. In their paper, the authors analyzed two software-based FIFO implementation strategies. In addition, they investigated how to use the heterogeneous memory structure efficiently. Key is the efficient packetizing of communication tokens. Their experimental results emphasize the importance of tailored FIFO channels. However, their analysis focuses only on execution time. PICO automatically explores several synchronization implementations to find the optimal configuration regarding several objectives, including run time and energy consumption.

Traditional FIFO channels introduce additional data copy overhead during read and write operations. Windowed FIFOs [HGT07] remove this overhead by introducing additional channel operations for sharing the data between concurrent tasks. In principle, concurrent processes do not have a local copy of the data, instead they work directly on the shared buffer. Thus, local operations on the shared data take place in the shared memory protected by index pointers to prevent collisions and guarantee a correct execution order. Haid et al. [HSH09] demonstrated that windowed FIFO-based communication in KPNs using protothreads can be efficiently executed on the Cell platform. Protothreads [DSV06] realize cooperative multi process execution on a single processor. Their evaluation only focuses on speed-up and does not consider other objectives important for embedded systems. In addition, if starting from a sequential application, the approach requires substantial work from the developer to transform the code.

An efficient KPN mapping to a target platform should not only consider the computation tasks but also the communication between the nodes. This is especially important in the case of heterogeneous systems which provide different communication resources. Castrillón Mazo et al. [CTL12] developed a Group-Based Mapping (GBM) heuristic using timing information obtained by tracing information and FIFO capacities provided by the MAPS tool [CLA13]. Section 3.3 provides a more detailed discussion regarding the parallelization aspects of the MAPS tool. Using a joint process, GBM assigns

communication and process mapping to minimize the execution time of the application. Therefore, the heuristic maps elements of the KPN to groups of hardware components first, and then maps the members of each group. All components of a group share similar capabilities like timing characteristics. This work highlights the importance of considering communication mapping to different resources as a vital part of the optimization process to reduce the execution time. In contrast, PICO is able to consider multiple communication techniques with respect to multiple objectives. The GBM approach has been refined by using a split-cost communication model by Odendahl et al. [OCV13]. In their work, they analyzed three different channel implementations and mapping to DRAM and scratch memory and observed that traditional single cost models to calculate the communication costs are not accurate enough. Therefore, they split the costs into a receiver and sender value. The enhanced GBM heuristic achieved an improvement of 10%. PICO's execution model also uses different costs for reading from and writing to a channel.

Ko and Won [KWB10] analyzed the impact of buffer sizes and synchronization performance on the overall system's performance of heterogeneous systems in context of image processing applications. Their model also considers different communication costs for sender and receiver. Their results emphasize the impact of buffer synchronization overhead especially for small buffer sizes.

Ferrandi et al. [FLP10] presented a combined ant colony optimization-based heuristic for mapping and scheduling of tasks and communication onto heterogeneous multiprocessor systems. Their multistage decision process approach decouples scheduling and mapping. The target platform is modeled using an abstract architecture model. The main objective is make-span but other important aspects like energy consumption are not considered. In contrast to PICO, their approach only considers a fixed type of communication implementation.

Erbas et al. [EEP03] proposed a combined task and FIFO mapping approach using a genetic algorithm. Applications are modeled as KPNs. Their multi-objective aware exploration approach is specifically tailored towards the Sesame framework. The approach considers mapping of channels but not how the channels are implemented. Hence, only one FIFO channel type is used. Verdoolaege's [VNS07] pn compiler derives process networks from Static Affine Nested Loop Programs (SANLP). For these specific loops the compiler can statically determine a safe lower bound for the FIFO capacity to guarantee a deadlock free execution. Section 3.3 discusses the parallelization aspects of the pn compiler and its usage in the MADNESS project.

Determining the optimal communication channel capacity with respect to multiple objectives under resource limitations is crucial. Therefore, the buffer sizing problem has been investigated in the last decades. In the process network domain, especially for Kahn Process Network, Parks [Par95] investigated bounds for buffer sizing theoretically. The proposed algorithm starts with a limited (small) buffer size and increases uniformly the capacity for all channels in case of a deadlock. Usually not all channels of the

network are involved in the deadlock. Thus, [GB03] extended Parks' algorithm so that only the capacity of FIFOs responsible for the deadlocks are increased. The presented works resolve the deadlock issue, however, since PICO starts with sequential applications the resulting parallel application do not suffer from deadlocks produced by wrong channel capacities. In our case, the FIFO capacity may influence the performance and therefore a good trade off between multiple objectives must be found. For rate constrained applications, Cheung et al. [CHB07] proposed an automatic off-line buffer sizing algorithm to find the minimal FIFO capacities to meet the application constraints. In an iterative approach their algorithm increases the FIFO sizes to improve the performance in terms of run time. However, as for many KPN-focused approaches they do not consider merging of multiple FIFO channels to increase the performance. Gogniat et al. [GAB98] presented a HW/SW communication synthesis and resource optimization approach. Their method explores different communication methods like bus-based or hardware FIFO data exchange. In addition, the algorithm merges communication resources and focuses on static applications.

Bus and shared memory-based communication infrastructures may not be suitable for future many core systems. For such systems, (hybrid) NoC interconnects might offer a good solution to connect large amounts of processors. Hu and Marculescu [JM04] presented a static scheduling heuristic for tile-based NoC architectures. Their algorithm takes energy consumption and real-time constraints during the mapping process into account. Further, the approach supports heterogeneous systems and therefore models different execution times for each task depending on which processor it is mapped to. The energy consumption model for transmitting data through the network is basically the sum of the consumed energy by the switch and link between network tiles. Lee and Choi [LC12] presented a combined task and communication mapping approach which considers shared memory and message passing to realize the data exchange in a homogeneous system. Their communication routing approach is able to optimize execution time or energy consumption. In contrast, PICO is multi-objective aware and considers all objectives simultaneously. Their many-core target architectures provide a NoC and memory communication infrastructure. The communication model of this approach takes communication delay into account which may come from network contention.

To conclude, various publications highlight the importance of the FIFO mapping onto the available resources. Mapping of communication is important but most methods do not consider the flexibility offered by various channel implementations. Most of the presented approaches only consider execution time as their main objective and thus neglect important aspects of resource-constrained embedded systems. In addition, FIFO channel merging, which might reduce overhead, is vastly neglected. In contrast, PICO provides an automatic multi-objective aware approach to optimize the communication in parallelized applications which considers multiple communication types, mapping, capacity and channel merging specifically targeting low-power resource-restricted embedded systems.

4.4 Internals of the Communication Optimization

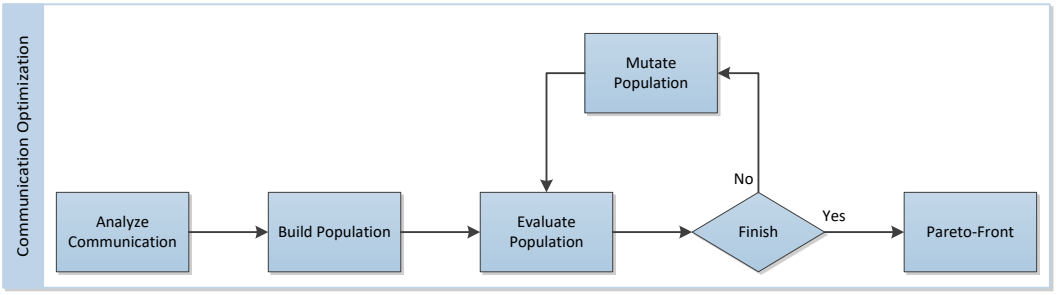


Figure 4.2: Communication optimization phase.

This section covers details regarding the communication optimization flow implemented in PICO. The optimization algorithm is tightly integrated into PICO's parallelization process. Figure 4.2 shows the internal steps of this phase. The optimization approach internally works on the parallel task graph presented in Subsection 3.6.1.3 following the application model presented in Subsection 3.2. The basic idea is that each data synchronization node provides several parameters building the exploration space for the optimization algorithm. For instance, the channel type identifier defines which specific synchronization implementation is to be used for that node. Additional parameters control the capacity and mapping of the FIFO to the memory. As stated above, PICO's communication optimization process is designed to be extensible and thus the user can add new communication methods to the design space. Accordingly, the user provides the software implementation and properties implementation parameters to PICO. In a future extension, the process could be automated in such a way that PICO performs an analysis to extract information necessary for the optimization process. Here, PICO generates a new identifier and calculates the memory overhead for this implementation. Later, the memory overhead in combination with the FIFO size determines if a certain solution can be mapped to a specific target platform.

PICO's GA implementation is based on the PISA framework [BLT03]. It takes care of the selection process and provides additional implementation support. PICO therefore is responsible for the chromosome structure, fitness evaluation, cross-over and mutation. The fitness evaluation typically dominates the run time of GA-based optimization algorithm. Therefore, we schedule multiple evaluations in parallel for a given generation, which significantly reduces optimization time. The following sections provide more detailed descriptions of the employed algorithms.

4.4.1 Genetic Algorithm Implementation

This section covers implementation details of PICO's GA-based optimization process. First we present the (extensible) chromosome structure used to encode all parameters of the problem space. Then we describe the cross-over and mutation operations performed by PICO during the optimization process. Finally, we discuss the fitness evaluation.

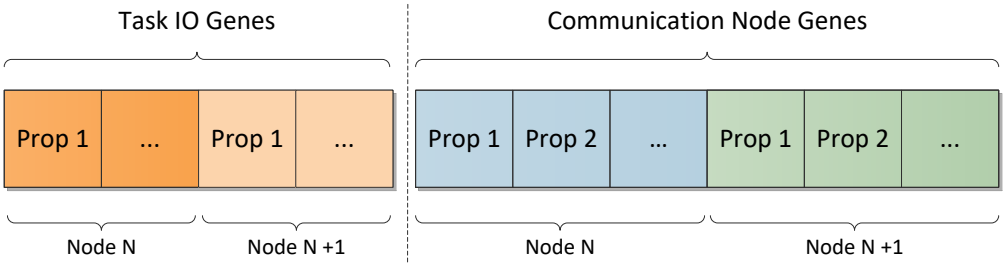


Figure 4.3: General chromosome structure.

4.4.1.1 General Chromosome Structure

The chromosome structure defines the mapping between the problem space (parameters) and the genetic optimization algorithm. One gene of the chromosome basically corresponds to one parameter, e.g., capacity. An expressive and evolvable representation is necessary to guarantee a good and robust optimization process. Internally, the chromosome is an array of bits with a variable length. PICO uses integer arrays where each element represents a different parameter. Thus, all parameters must be encoded into this array representation.

General Chromosome Structure: The application model used in the PA4RES framework (cf. Subsection 3.2) and the resulting internal representation generated by PICO (cf. Subsection 3.6.1.3) contain two types of synchronization nodes. Task in and out nodes synchronize data between parent and child tasks and they are usually executed only once. Thus, a channel-based synchronization only makes sense for specific corner cases. Nodes synchronizing data between concurrently running tasks are processed multiple times and thus use channel-based synchronization. Figure 4.3 visualizes the general chromosome structure with respect to the two separate communication node types. The first part (Task IO Genes) of a chromosome is reserved for *taskIn* and *taskOut* node parameters and the second part is for FIFO communication nodes. Each graph node can have several parameters (genes). In this approach, synchronization from the parent task to a child can be implemented different to the communication from the child to the parent. Thus each *taskIn* and *taskOut* node is represented in the chromosome, whereas data exchange between concurrently running tasks is specified by the *comOut*. Accordingly, the chromosome only contains half of the communication nodes. The length of the chromosome is variable but fixed for a specific parallel application which relates to the number of synchronization nodes multiplied by the number of parameters. Our general chromosome layout allows us to specify an arbitrary parameter set nested in the nodes.

Specialized Chromosome Structure: For this chapter, we use the simulation-based platform as a testbed to evaluate the GA-based optimization algorithm. Inheriting the general structure, Figure 4.4 shows the chromosome used to optimize data synchronization for the simulation-based target platform in this thesis. This system provides a large but

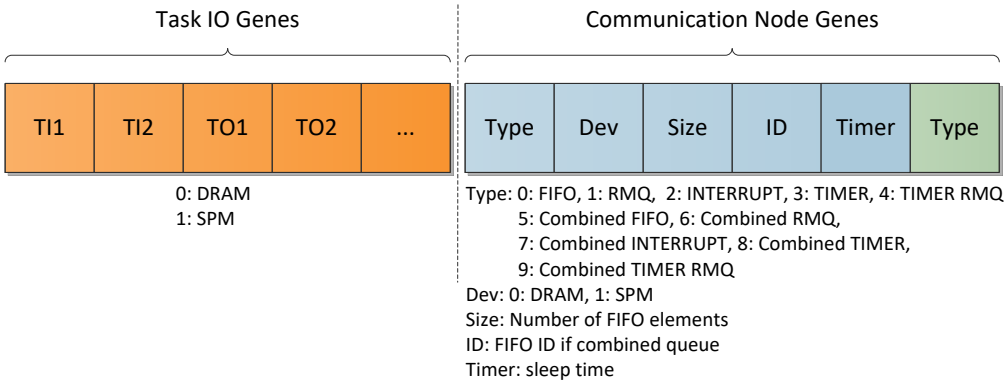


Figure 4.4: Chromosome structure used in this thesis.

slow and energy inefficient DRAM and a fast, energy efficient but rather small SPM which can be used for data exchange between the processors. The mapping to the memory is encoded using a *Dev* gene, where the value 0 represents the DRAM and 1 the SPM. In this case, each task IO node and com node have this gene. Further, we implemented five different software FIFO channel implementations which should be explored by the GA. The *Type* gene encodes the communication node to implementation mapping. Type 0 is the standard FIFO implementation based on the busy waiting principle. If no data is available or the channel is saturated and the access blocked, a loop constantly checks for data or space availability. This implementation tends to reduce the waiting delay since it proceeds as soon as possible but wastes energy for the polling mechanism. The implementation of type 1 uses the queue system provided by the RTEMS operating system. It can only be used for very small data (380 bytes). In contrast to type 0, type 2 uses an interrupt-based approach. In the case of blocking, this implementation sets the processor to a sleep mode to reduce the energy consumption. The sleeping task wakes up on each system interrupt and checks if the channel is still blocked. Thus, the execution time might increase due to the suspension but the energy savings heavily depend on the amount of interrupts. Finally, type 3 and 4 use a configurable sleep timer where 4 uses the RTEMS queue system. Here, the GA configures the sleep period to a fixed value. In contrast to the interrupt-based approach, this implementation might benefit more from an undisturbed suspension. For each FIFO type we allowed PICO to implement a combined version, in this case, they are numbered from 5 to 9 where type 5 is a combined polling-based queue and 9 a combined sleep timer-based RTEMS queue version.

For completeness, Figure 4.5 shows the structure used in [NEM15b]. In that version, only 4 different channel types were supported. Most notably in comparison to the structure used in [NEM15b], the structure used in this thesis provides combined channels for all FIFO types.

4.4.1.2 Genetic Operations

During the iterative optimization process, the GA performs several genetic operations to create a new generation from an existing population. Therefore, the algorithm conducts

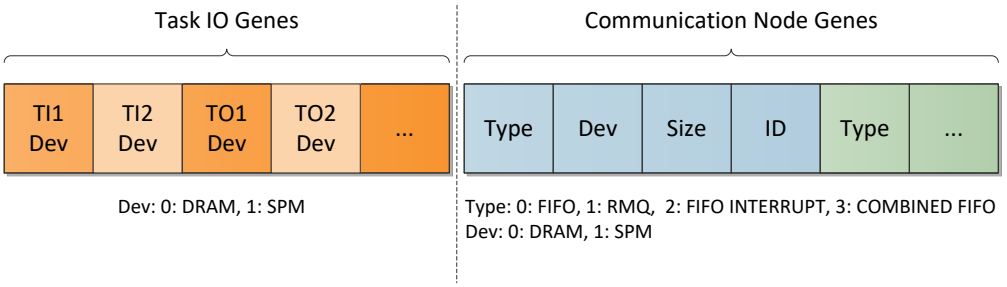


Figure 4.5: Chromosome structure used in [NEM15b].

cross-over and mutation operations on the individuals inspired by nature to form a new generation. These operations might result in an invalid individual, for instance a solution exceeding the available memory with the newly generated channel capacity. Thus, our GA-based optimization algorithm repairs those individuals.

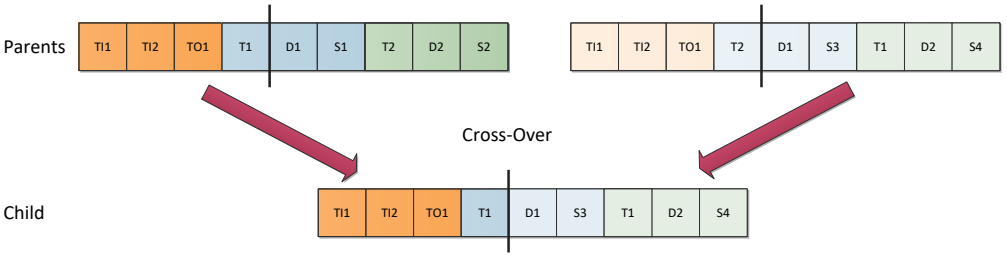


Figure 4.6: GA cross-over operation: merge two individuals at a random position.

Cross-over: Figure 4.6 visualizes the cross-over operation which combines two individuals into a new solution. Our chromosome has a variable length depending on the number of *taskIO* and *comOut* nodes. However, the length is fixed during the optimization process since the parallel application and thus the graph do not change. PICO's cross-over operation randomly selects a gene position for the recombination. For the new individual, genes located before this cross-over position stem from the first original individual whereas genes located after that position originate from the second individual. This combination might result in an invalid solution. In that case a repair process steps in.

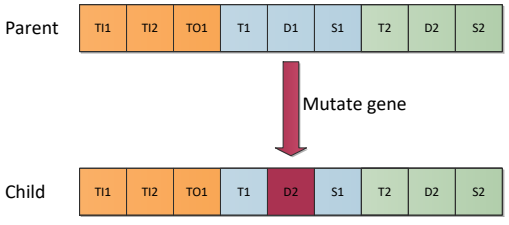


Figure 4.7: GA mutate operation: select random position and change value randomly.

Mutation: Genetic mutation changes the values of genes. Figure 4.7 illustrates this operation. We implemented single and multi bit mutation. For the single bit mutation,

the algorithm selects one gene randomly. Whereas, in the case of multi bit mutation, the GA-based approach selects several genes randomly. Then, the algorithm randomly assigns new values to the genes respecting the value range of this gene. This additional step does not prevent that the mutation produces invalid individuals. Hence, an additional repair step is necessary.

Repair: The cross-over or mutation process might produce invalid individuals, thus our GA applies an additional repair step. In classic GA these individuals get sorted out, but to improve the overall optimization time, we apply a repair step to generate a valid solution instead of to timely execute a faulty individual. An invalid solution might be an impossible combination of channels. In such a case, the repair function reverts the combination of channels back to separate synchronization. A mismatch between the FIFO types used by the nodes combined into a channel is solved in the way that the lowest FIFO type number is used for all channels assigned to this combined channel. Similar, if the capacity varies among the channels which should be combined, the repair step sets the capacity to the smallest size used by the involved channels. Another issue is the memory consumption of a channel configuration. It might be, that a specific software implementation can not handle the required capacity or resulting size of a combined FIFO. In such a case, the algorithm tries to lower the number of elements or split the combined channel. If the resulting individual is still invalid, the following fitness evaluation will assign a bad fitness value to this individual and it is very unlikely that this individual will survive the selection process.

4.4.1.3 Fitness evaluation

The fitness evaluation is an essential part of the optimization process. It assigns a fitness value (performance) to each individual to steer the evaluation process. Individuals with a high fitness should survive and reproduce and those with lower fitness should vanish. This optimization process is inspired by nature and mimics survival of the fittest. A fast but robust fitness evaluation is therefore vital since GA algorithms may evaluate millions of individuals. We extended the PISA implementation and enabled our GA to evaluate all individuals of a generation in parallel. This reduces the evaluation time drastically under the permission that sufficient target systems, simulator instances or computation capacity are available. To further speed the evaluation process up, a database stores all already analyzed solutions and their fitness value to avoid re-evaluation. Some parameters can be evaluated statically which may avoid a possible expensive execution-based evaluation. The memory consumption of the FIFO-channels of an individual can be calculated statically. To determine the execution time and energy consumption, the GA either executes an individual on the target platform or consults a high-level execution model. A hybrid fitness evaluation model which most of the time prefers the fast model-based calculation but sometimes a precise detailed evaluation run might be a worthwhile

extension. In the following we will provide more details on the objectives and evaluation methods.

Objectives: In this chapter, we consider PICO's fitness evaluation of *run time*, *energy consumption* and *memory consumption* for all memories separately. The execution time and energy consumption can either be measured or estimated using a high-level execution model (cf. Subsection 4.4.2). In case of the memory consumption, the fitness values can be calculated beforehand and used to sort invalid solutions out to prevent time-consuming (faulty) evaluations. Therefore, the resulting Pareto-frontier evolves toward short execution times, low energy consumptions and small memory footprints.

Static Memory Consumption Evaluation: PICO calculates the memory requirements for the communication of the individual solution before a potential expensive long-running measurement. In the following, we provide all necessary equations used by PICO to evaluate an individual regarding memory consumption and feasibility of the solution with respect to memory restrictions. In general, the memory occupied (MEM) by a (software) communication implementation for a given memory m is the sum of the space occupied by the single and combined FIFO channels assigned to that memory.

$$MEM(m) = SCOM(m) + CCOM(m) \tag{4.1}$$

The amount of memory space used by single FIFO channels ($SCOM$) is the sum over all single channels sizes. For each channel, the size depends on a type-dependent overhead and the bytes required to store the data buffer.

$$SCOM(m) = \sum_{n \in COM(m)} Overhead(n) + Bytes(n) \tag{4.2}$$

In case of a combined channel, the type-dependent overhead only is added once for each combined FIFO. The memory required to store the data is equivalent to the single FIFO. Therefore, the size required to implement all combined channels ($CCOM$) for a given memory m is:

$$CCOM(m) = \sum_{comb \in COMBINED(m)} (Overhead(comb) + \sum_{n \in comb(m)} Bytes(n)) \tag{4.3}$$

The memory space reserved for the data is the FIFO capacity multiplied by the size in bytes for one element of the channel:

$$Bytes(n) = Capacity(n) * SizeOfElement(n) \tag{4.4}$$

Finally, memory space is limited either physically or artificially restricted by the user. Thus, we must ensure that only solutions survive which meet these conditions:

$$\forall m \in Memories : MEM(m) \leq MAXSIZE(m) \tag{4.5}$$

In case of the simulation-based target system, the channels can be mapped to two types of memory. The SPM is rather small and therefore, an individual might generate a channel mapping which exceeds the available capacity. In such a case, the solution is invalid resulting in a low fitness and no expensive simulation is necessary.

Online Run Time and Energy Evaluation: The GA interfaces with the simulation-based and with the real hardware platform. The PA4RES framework provides several internal methods to execute code on the target platform. In case of the simulation-based systems, the overall execution time as well as the active time of each processor is measured by the simulator itself and reported to the algorithm. The framework adds start measurement instructions at the beginning of the application so it excludes operating system boot time. In addition, it adds proper end measurement instructions to the end of the application which should be optimized. The energy consumption is calculated using the EnergyMetric (cf. Section A.1) module and returned to the optimization framework.

In case of the Odroid-XU3, the GA connects via a SSH tunnel to the board. Here, it can either use the EnergyMeter or the Energy Relay Reader (cf. Section A.2) to measure the energy consumption and execution time. To improve the precision of the results, the GA is configurable to repeat these measurements. The results are then reported back to the optimization algorithm. In either case, the user of PICO can configure a timeout to stop the execution of unwanted long running individuals to speed up the evaluation process. In this case, the evolution assigns a poor fitness value to these terminated individuals.

4.4.2 Execution Model

Simulation-based exploration can be very time-consuming. Therefore, PICO provides a high-level execution model, capturing the essentials of the parallel application and target platform. This model is tailored towards a fast evaluation of the communication optimization task and abstracts several aspects of the application. The execution model consists of a communication graph *CG* and a target architecture model. The aim of this model is to provide guidelines regarding the FIFO channel configuration and mapping. The model is not meant to provide a precise performance evaluation of the entire application. It is precise enough to compare different solutions on an abstract ratio basis. Ideally, the model should be used for a rapid search space pruning followed by a detailed simulation-based exploration or in a hybrid approach.

The key idea of the model is, that it abstracts computation away such that it is represented by a fixed cost, e.g., execution time or energy consumption. Instead of data, the communicating tasks now exchange timestamps for FIFO channel access. With this information, we can model the blocking time of channels appropriately. To model the interaction between concurrently running tasks, PICO builds a Communication Graph:

Definition 4.2 (Communication Graph (CG)):
A communication graph is a tuple (P,C) of processing (P) and communication (C) nodes. Nodes are connected with edges that represent the control flow and define the execution order. Communication nodes are either of type sending or receiving and model indirectly the data dependency between parallel tasks.

To consider the specific capabilities, PICO uses an architecture model:

Definition 4.3 (Architecture Model):
An architecture model represents the available processing and communication capabilities of the target platform. It provides estimated processing and communication costs. In addition, it models which communication infrastructure can be simultaneously used.

Finally, we can define the execution model as:

Definition 4.4 (PICO Execution Model):
The execution model traverses the CG with information provided by the architecture model and accumulates the costs of each node. In the case of blocking, the model takes waiting time and FIFO configurations into account.

Execution Model Construction: To reduce the evaluation time, the CG should only capture the essentials of the parallel application in context of PICO's communication optimization. Hence, the original application is reduced to processing and communication nodes. Figure 4.8 visualizes this process briefly. Depending on the mapping, PICO can select the correct values for each task from the architecture model. Therefore, PICO enriches the task graph (cf. Subsection 3.6.1.3) with data provided by the Performance Estimator (cf. Subsection 2.3.1). The data consists of run time and energy values for each statement class mapped to all available processors.

In the next step, PICO folds all execution nodes between communication nodes into a single processing node. Hence, parts without parallelism or communication are not considered and removed from the graph. It calculates the longest path between two communication nodes of the same task. To find the longest path, PICO's heuristic traverses the graph in a bottom-up way. Thus, the inner most nested statement nodes are visited first and their values are summed up. Once PICO reaches a split in the control flow, e.g. at a loop head or a conditional statement it performs a merge operation. In case of a loop, PICO multiplies the accumulated values of the nested nodes by a static loop count. The loop count is either known or derived from *flow facts*. In the case of a conditional statement, PICO selects the most expensive path. If a parallel region calls a function, PICO computes the high-level costs of this function beforehand and stores this value for a faster lookup. If a called function is too complex for the abstract model, PICO may require additional user input. The high-level execution model currently does not support nested parallelism. In the worst case PICO fails to generate a model and reverts to the simulation-based evaluation for the communication optimization. However, since the task graph is well-structured, loop bounds are known and the input language is

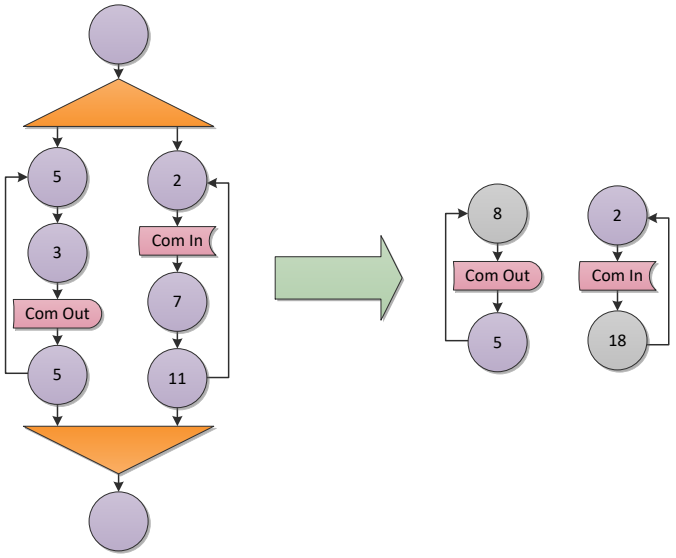


Figure 4.8: Execution model visualization, numbers inside nodes represent costs.

restricted to avoid side effects, PICO is able to approximate the longest path is in most scenarios used in this thesis.

Modeling the communication including delay and contention is vital for a sufficiently accurate performance estimation. At this stage, the GA provisioned the channels, so that channel capacity, mapping and implementation details are known. In this model, the communication cost is split into a static and a dynamic part. The static part accounts for the case of non blocking access to the channel whereas the dynamic part represents the blocking time. Sending and receiving may not cost the same as the model provides separate costs for these nodes. The model provides byte-based transmission costs and delays for each of the available FIFO implementations. These values have been manually measured and calculated for all currently available FIFO types. In addition, the model considers the data transmission costs between parent and child task. In the future, this process can easily be automated using predefined measurement routines. The dynamic communication and overall execution costs are estimated during the execution of the model. In the following, we provide a detailed description of the actual model execution and how the dynamic costs are derived.

Model Execution: To estimate the impact of the communication setting, PICO executes the model by traversing communication graph for each parallel region. The employed approach is comparable to virtual loop unrolling and mimics the execution of the entire section without actual data. Therefore, PICO forks a concurrent (Linux) thread on the host for each parallel task. These threads *execute* the folded task graph and simulate the communication in the graph. The model uses a global time and energy consumption model. In the case of heterogeneous architectures, the run times are normalized based on the lowest common frequency. Executing a processing node is simply adding up the processing time values. Modeling the communication is sophisticated.

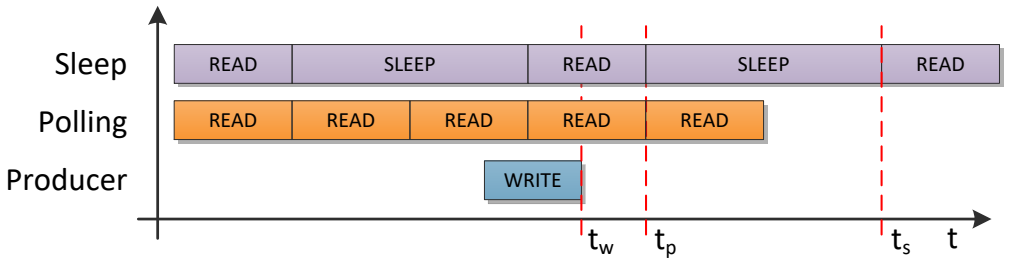


Figure 4.9: Communication waiting cost modeling.

Instead of actual data, the channels exchange timestamps for read and write accesses. This enables the model to account waiting for blocked access. Algorithm 4.4 provides a pseudo code representation of the key components of the communication modeling.

PICO's execution model internally represents a FIFO with a *channel class* to model the blocking nature of a FIFO. This class has two queues, one storing the write times q_w to that channel and one the read times q_r. In addition the class also stores the capacity of the FIFO. Further, this class provides read and write methods.

Channel Access Modeling: In case a communication node sends data, the write method is called with the current start time t_s and the duration d (costs) of a non blocked write process for this specific FIFO type. The write method first checks if the FIFO capacity is sufficient to allow a write. If space is available, the write method proceeds and stores the time stamp the write completes $t_w + d$ into q_w. Finally, it returns the start time of the write process (t_s) to the caller.

A read works similar as the read method checks if q_w contains an entry. If data was written to that queue and since the PICO application model uses point-to-point channels, the read is successful. Then, the read method removes that entry from q_w and adds the current read time $t_r + d$ to q_r.

In case of blocking access, PICO must approximate the waiting time properly. At this point, the contents of q_w and q_s play an important role. A blocking write means that q_w is full. Therefore, the execution of this task blocks and waits until a read on this channel occurs which removes one element in q_w and the write method proceeds. To model the waiting time, the write method now uses the time t_r that the previous read stored in q_r as the starting point for the write. Therefore the write method returns $t_r + d$ to the caller.

Communication Cost Modeling: To model the energy consumption and account for sleep modes for that FIFO access, the model calculates the time span t_{delay} between the time passed to the method and the returned value. Each FIFO type has a specific cost in terms of time and energy for a single waiting event. This models for instance various sleep modes.

Figure 4.9 visualizes this. Let us assume, a read starts at time 0 but the write at time $t_w = 150$. In case of a polling FIFO, we would constantly try to read the channel.

Algorithm 4.4 Communication Model

procedure *Channel::write*(*startTime, duration*)
  $writeTime \leftarrow startTime$
  pop *waitTime* ***from*** q_r
  if $size(q_w) + 1 \leq capacity$ **and** $waitTime \leq startTime$ **then**
    push *startTime + duration* ***to*** q_w
    pop *elem* ***from*** q_r
  else
    while $size(q_w) + 1 \geq capacity$ **do** *wait()*
    push *waitTime + duration* ***to*** q_w
    $writeTime \leftarrow waitTime$
  return $writeTime$
procedure *Channel::read*(*duration*)
  while $empty(q_w)$ **do** *wait()*
  pop *readTime* ***from*** q_w
  push *readTime + duration* ***to*** q_r
  return $readTime$
procedure *ComOutNode::execute*(*procID, globalTime*)
  $costs \leftarrow singleWriteCosts()$
  $duration \leftarrow getWriteDuration(procID)$
  $writeTime \leftarrow Channel :: write(globalTime, duration)$
  if $writeTime \neq globalTime$ **then**
    $diff \leftarrow globalTime - writeTime$
    $numberOfWaits \leftarrow \lceil diff \div singleWaitingTime() \rceil$
    $costs \leftarrow costs + numberOfWaits \times singleWaitingCosts()$
  return $costs$
procedure *ComInNode::execute*(*procID, globalTime*)
  $costs \leftarrow singleReadCosts()$
  $duration \leftarrow getReadDuration(procID)$
  $readTime \leftarrow Channel :: read(duration)$
  if $readTime \neq globalTime$ **then**
    $diff \leftarrow globalTime - readTime$
    $numberOfWaits \leftarrow \lceil diff \div singleWaitingTime() \rceil$
    $costs \leftarrow costs + numberOfWaits \times singleWaitingCosts()$
  return $costs$

Therefore, the read could read the channel successfully at t_p. Let us now assume a wait FIFO sleeps 100 time units between each check if data can be written. Therefore, the write could only proceed after $t_s = 200$ time units and we must add twice the waiting costs. Thus, $t_p \leq t_s$ but due to sleep state, $energy(t_s) \leq energy(t_p)$. The model accounts for that and the overall execution and energy consumption of a communication node is can be approximated with:

$$ComCost(node) = d + numberOfWaits * singleWaitCost \tag{4.6}$$

In the context of PICO, this fairly abstract model works since we can make some assumptions. The original application was sequential and its transformation rules into

a parallel application are known. In case of a combined channel, PICO only adds the costs once. Currently this model does not consider contention on the communication infrastructure. However, this could be added easily with an additional queue modeling the contention.

Total Cost Calculation: Finally, we estimate the final performance according to Equation 4.7 and Equation 4.8 The model accumulates the longest of the estimated execution time for the tasks in each parallel region. This models the fact that the control flow joins at the end of a parallel region and the execution can only proceed if all tasks of that region have been terminated. However, for the estimated energy consumption we must account all concurrently running tasks for each region. For nested regions, PICO could calculate the estimated performance in a bottom-up approach.

$$executionTime_{est} = \sum_{region \in parallelRegion} longestTaskTime(region) \tag{4.7}$$

$$enegyConsumption_{est} = \sum_{region \in parallelRegion} \sum_{task \in Tasks} getEnergyEst(task) \tag{4.8}$$

4.5 Evaluation

This evaluation section presents results regarding the communication optimization provided by PICO and particularly answers the following questions. Do the test case applications benefit from PICO's communication optimization? Is PICO able to explore the available channel implementations and various parameters efficiently? How does the execution model perform and how can it improve the optimization process?

4.5.1 Evaluation Setup

For this evaluation we conducted experiments with various real world and synthetic applications. PICO optimizes the necessary data synchronization within these test cases towards the objectives run time, energy and memory consumption, targeting a homogeneous and a heterogeneous multi-core platform. The optimizing phase uses a genetic algorithm to perform the parameter exploration. All evaluation runs were performed on a dual Intel Xeon CPU X5650 (6 cores, 12 threads) server with 54 GB memory. Parallel evaluation was utilized as much as possible but is limited by the available licenses for the simulator.

Figure 4.10 visualizes the overall optimization flow. The sequential application with PICO directives is passed the PICO which then constructs the application model graph. PICO then creates the chromosome structure according to that graph and starts the optimization process. Finally, PICO returns a set of Pareto-optimal solutions.

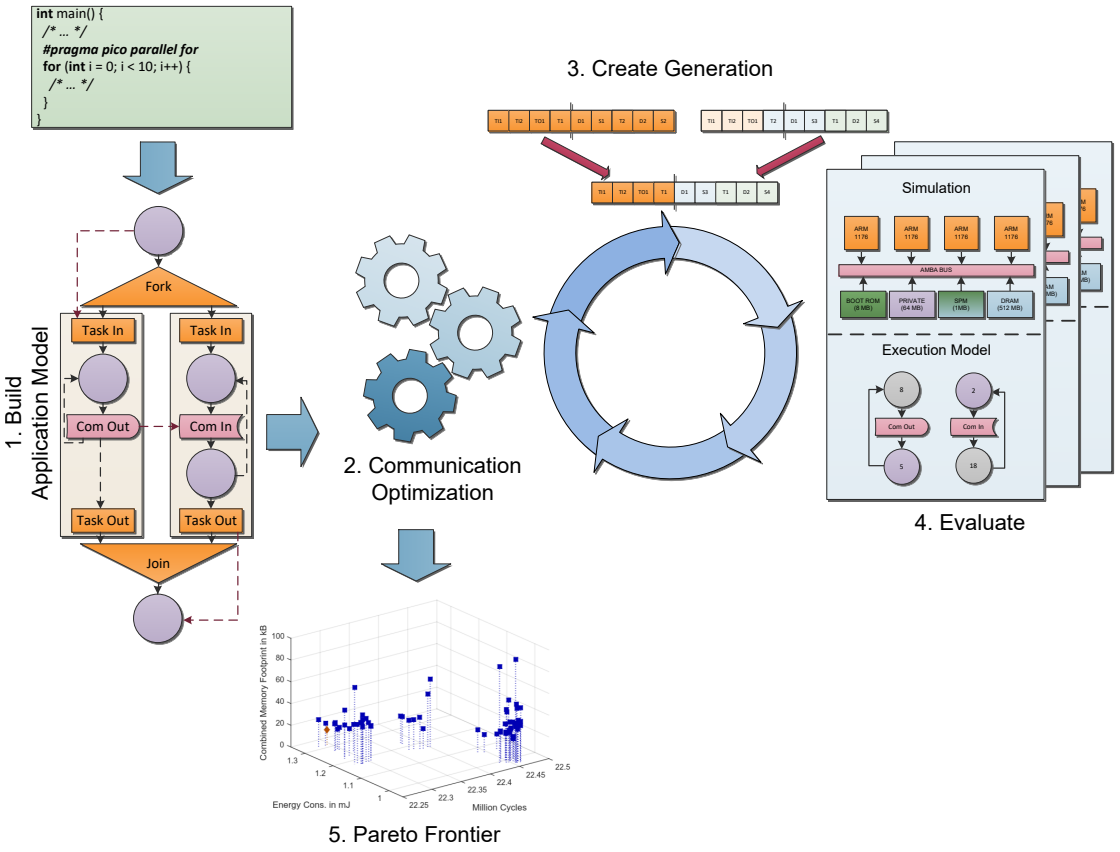


Figure 4.10: Communication optimization flow visualization.

4.5.1.1 Applications

To analyze PICO's capabilities we selected several real world applications and generated additional synthetic test cases. Table 4.1 summarizes the benchmarks used in the evaluation. The real work applications originate from the UTDSP benchmark suite [Lee08], the SNU real-time benchmark suite [SNU17]. The PAMONO preprocessing test case applies several image processing steps (cf. Section 3.7 or Section 5.4). These tests were also used in previous experiments and Section 3.7 provides additional insights. We selected these applications since they require data exchange between pipeline stages and thus should benefit from PICO's communication optimization. The other programs used in Section 3.7 might only benefit from optimized *taskIn* and *taskOut* communication and were excluded from this evaluation. As with the evaluation in Section 3.7, we used the parallelized versions of the application taken from the Pareto frontier generated by PAXES. In this case, we selected proper solutions for the target hardware platforms. For this evaluation, we selected a `PAMONO preprocessing` parallelization which uses FIFO channels.

Synthetic Pipeline Test Cases: To gain additional insights into PICO's capabilities we apply PICO's communication optimization to a set of synthetic benchmarks. With these artificial applications we inspect different aspects of the optimization algorithm and

Benchmark	Channels	Description
Filterbank	3/8	Pipeline of filter stages, including convolution, down and up sampling
Spectral Analysis	6	Calculates the power spectral estimate of speech using periodogram averaging
JPEG	3	JPEG encoder
PAMONO Preprocessing	10	PAMONO preprocessing
Synthetic Pipeline	10	Several synthetic pipeline test cases to explore PICO's capabilities

Table 4.1: Benchmark description - real world benchmarks taken from UTDSP [Lee08] and SNU benchmark suite [SNU17] and self-generated synthetic test cases.

```
#pragma pico parallel pipeline for num_threads(NUMTHREADS)
for ( i = 0; i < LOOP; i++)
{
  #pragma pico section T1SLICE
  { // Task 1
    for( j = 0; j < T1DUTYCOUNTER; j++)
    {
    in1 = 1; // Random write to variable
    }

    // ...

    for( j = 0; j < T1DUTYCOUNTER; j++)
    {
      in10 = 10; // Random write to variable
    }
  } // TASK 1
  #pragma pico section T2SLICE
  { // Task 2
    // define local variables
    // ...
    for( j = 0; j < T2DUTYCOUNTER; j++)
    {
      lin1 = in1; // force read from Task 1
    }

    // ...

    for( j = 0; j < T2DUTYCOUNTER; j++)
    {
      lin10 = in10; // force read from Task 1
    }
    result = lin1 + ... + lin10;
  } // TASK 2
} // LOOP
```

Listing 4.1: Synthetic pipeline test case template, configurable parallelism and complexity.

expect to be able to derive some general rules. We followed a template (cf. Listing 4.1) to derive the applications. A main loop is split into to two pipeline stages. Configurable loops are used to simulate workload for each stage. The template also provides hybrid pipeline parallelism. All test cases synchronize 10 variables across pipeline stage boundaries. However, we vary the type and size of the variables. Furthermore, the template provides a mechanism to leverage hybrid pipeline parallelism. To analyze the impact of channel merging, we modified the receiving part of the template thus all data is required at the same time. For another experiment we changed to order of the read, thus the first variable written will be read last. We set the outer loop to an iteration count of 20 and varied the synthetic work of the two stages: (1000/1000), (1000/100), (1000/10), (100/1000) and (10/1000). As data types, we used single integers and a slightly modified version using integer arrays with 64 elements. All synthetic experiments use the homogeneous platform with a SPM restriction of 512 B.

4.5.1.2 Target System

The evolutionary optimization algorithm employed in PICO explores versatile communication implementation and mapping-related parameters. Communication channel mapping is only suitable for systems with multiple memories ideally with different performance characteristics. Therefore, we focus on the simulation-based systems for this evaluation. Subsection 2.1.1 provides more details regarding these systems. Important for this section are the available memories. In this setup, PICO uses either the large, slow and power-inefficient DRAM or the small, fast and energy-efficient SPM for data exchange between concurrently running parts of the application. In our system, the operating system roughly consumes 400 KB of the 1 MB SPM, the remaining space can be freely utilized. PICO can be restricted to only assign certain amounts of memory. This might be beneficial for manually mapped variables or during prototyping to estimate the memory requirements for a later hardware product. To analyze how PICO deals with such limits we varied these memory restrictions in the experiment. Initially, in context of the considered applications and to stress the resource-restrictions faced by such systems we limited the available SPM to 512 B, 1024 B and 2048 B. Later, we extended the SPM to 32 KB for the `Filterbank` test case and 8 KB for `Spectral`. The FIFO management structures and PICO runtime variables are always assigned to the SPM. The four cores for the homogeneous system run with 500 MHz and on the heterogeneous one they are clocked with 500 MHz, 500 MHz, 250 MHz and 100 MHz, respectively.

4.5.1.3 Genetic Algorithm Configurations

PICO employs the SPEA2 [ZLT01] evolutionary algorithm provided by the PISA framework [BLT03] for the parameter exploration to find the true or an approximation of the Pareto frontier. In this evaluation, the fitness function evaluates an individual solution regarding run time, energy and memory consumption of the SPM and DRAM. The mutation property is set to 0.5 and a generation comprises 100 individuals. We let

the GA evaluate 100 generations. In case of an invalid or already evaluated solution, the optimization algorithm performs up to 10 additional genetic operations to generate a new unknown configuration. Therefore, the total number of evaluated individuals varies across the experiments. As objectives, the GA considers execution time, energy consumption, DRAM and SPM footprint. The GA minimizes energy consumption and execution time whereas SPM usage is maximized since this should improve the energy consumption. For this evaluation, we did not use an early termination, however, PICO provides such feature.

4.5.2 Simulation Results

Table 4.2 lists the number of simulations, genetic operations and the evaluation time for the benchmarks. In general, we observed that for cases were the data which must be synchronized is close to or exceeds the SPM restrictions, fewer simulations were performed. In such a case, the GA was unable to find new promising individuals which might improve the overall performance. Therefore, we extended the SPM limits for some benchmarks. The GA performs a simulation for all valid unseen solutions. In case of an invalid or known solution, PICO performs additional genetic operations to create a new individual. This explains the high number of GA operations. Overall, the evaluation time heavily depends on the execution time of a single individual and varied between a few hours to more than a day with up to 10 parallel evaluations. In the following, we present the results for the experiments. The plots contain execution time in cycles, energy consumption and the combined memory footprint. It is important to take into account that the GA considers SPM and DRAM utilization separately, but we chose, for better visualization, to report a combined value. In the following, we present an excerpt of the results for a better readability, the remaining result graphs are move to Appendix B. We compare our results against a baseline experiment, which uses a predefined FIFO mapping, channel size and type. In principle, this baseline represents roughly how the PAXES parallelizer internally accounts for data synchronization. This baseline experiment is visualized with a red diamond.

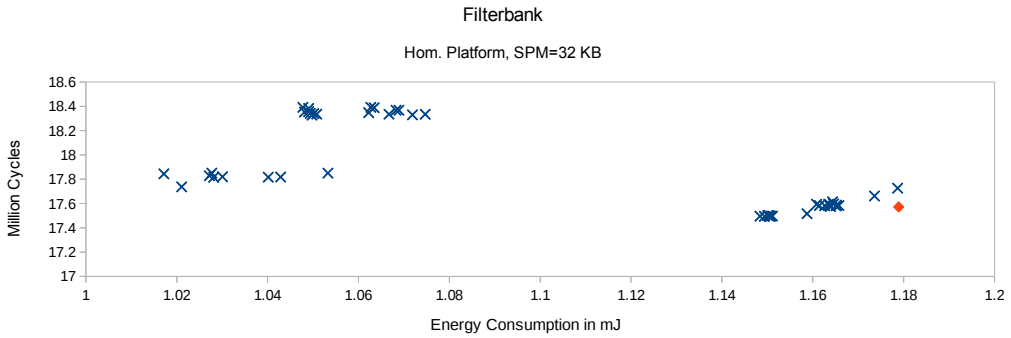
Filterbank: Figure 4.11 and Figure B.1 to Figure B.3 in Appendix B and show the results of the `Filterbank` experiments on the homogeneous system. This benchmark synchronizes a comparably large amount of data leading to similar results for a SPM of up to 2 KB. In such case, the GA mainly explored task in and out mappings as well as implementation types. Therefore, we extended the restriction to 32 KB. Overall, the results show solutions are grouped into two islands. The left, energy efficient, island used more sleep- and interrupt-based FIFOs whereas the other used the polling-based FIFO implementation more. This proves the intuition that waiting is more energy efficient but slower than polling. For this setting, the results do not contain solutions with merged FIFO channels or RTEMS queues.

Benchmark	SPM (B)	Number of Simulations	Number of GA Operations	Evaluation Time (h)
Filterbank (hom)	512	386	46,603	4.0
Filterbank (hom)	1,024	394	47,513	4.2
Filterbank (hom)	2,048	437	46,294	4.2
Filterbank (hom)	32,768	4,362	56,326	19.3
Filterbank (het)	512	250	48,552	3.4
Filterbank (het)	1,024	252	48,620	3.6
Filterbank (het)	2,048	447	52,035	4.2
Filterbank (het)	32,768	3,823	55,550	17.5
JPEG (hom)	512	2,009	46,353	20.4
JPEG (hom)	1,024	2,391	46,976	22.1
JPEG (hom)	2,048	3,461	46,484	26.2
JPEG (het)	512	1,917	45,784	19.6
JPEG (het)	1,024	2,522	47,844	23.8
JPEG (het)	2,048	3,117	49,332	24.1
Spectral (hom)	512	414	43,374	2.5
Spectral (hom)	1,024	913	48,629	3.6
Spectral (hom)	2,048	1,681	45,971	7.3
Spectral (hom)	8,192	2,899	47,478	8.0
Spectral (het)	512	435	45,182	2.5
Spectral (het)	1,024	1,338	46,605	4.6
Spectral (het)	2,048	1,976	41,140	8.1
Spectral (het)	8,192	3,042	46,668	8.8
PAMONO (hom)	512	603	42,685	48.3
PAMONO (hom)	1,024	958	41,120	68.7
PAMONO (hom)	2,048	1,092	43,334	75.6

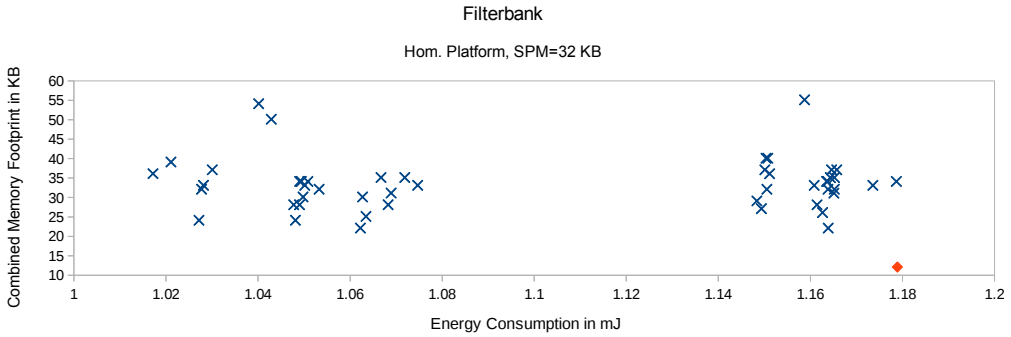
Table 4.2: GA statistics for communication optimization.

Leveraging the SPM restriction to 32 KB enabled the GA to explore the SPM for the data synchronization. In this case, the results show that in almost all cases the FIFO channels were mapped onto the energy efficient SPM. This led to an additional improvement compared to the other settings. The combined memory footprint indicates that for this benchmark, the channel size is not an important parameter. We expect that the overall memory consumption could be reduced if we would adjust the repai function such that more valid solutions are generated.

The same observation can be made for the heterogeneous platform. Figure 4.12 visualizes the results for this platform. The remaining results for this platform can be find in Figures B.4 to B.6 in Appendix B. Also, the results indicate that the solutions are more sensible to which waiting strategy is used. Overall, the results show, that PICO is able to explore the solution space and generate a broad range of solutions. Compared to the baseline result, fast and energy efficient solutions with a reduction of 30% could be found.

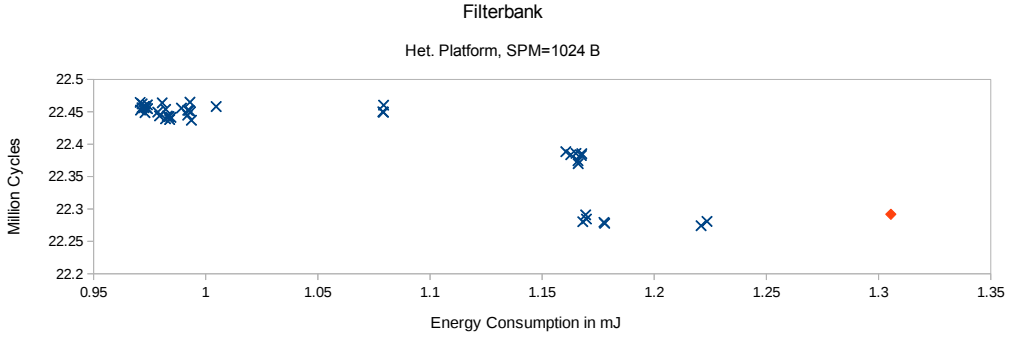


(a) Run time and energy consumption

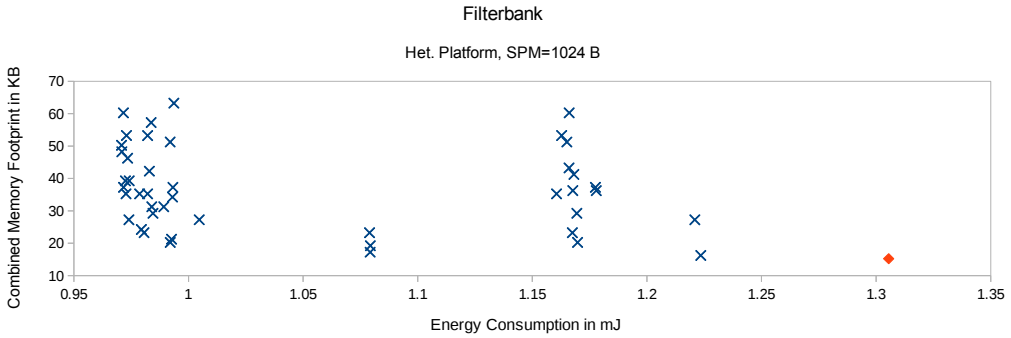


(b) Combined memory footprint and energy consumption

Figure 4.11: Filterbank on the homogeneous system with SPM restricted to 32 KB.



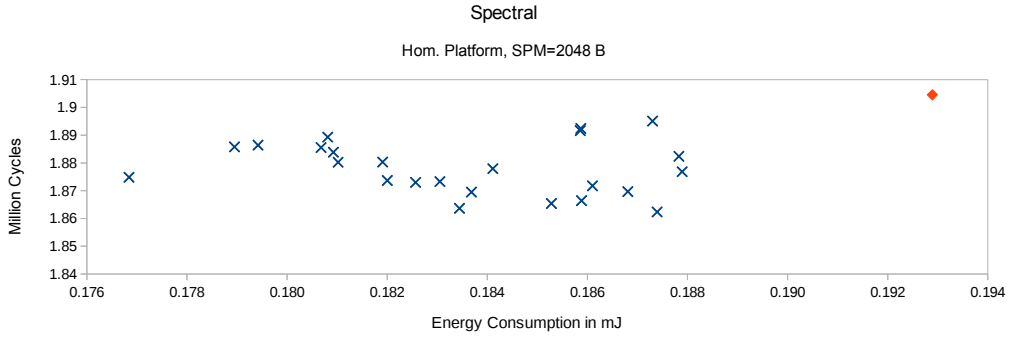
(a) Run time and energy consumption



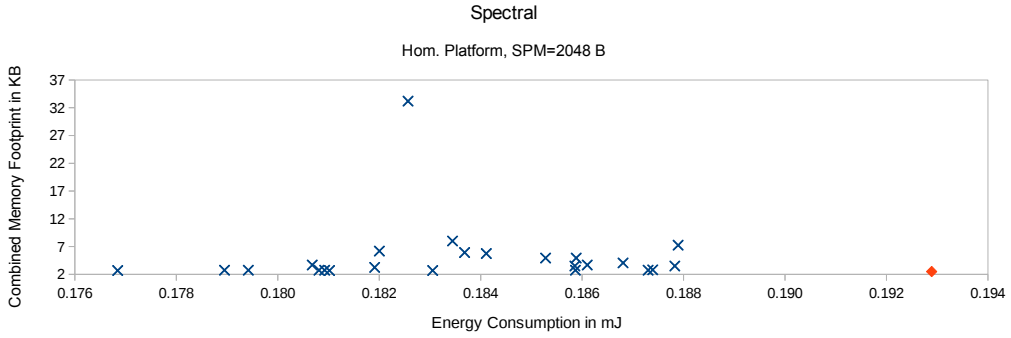
(b) Combined memory footprint and energy consumption

Figure 4.12: Filterbank on the heterogeneous system with SPM restricted to 1024 B.

Spectral: The `Spectral` benchmark suffers from strong data dependencies. As Section 3.7 shows, already the improvements due to parallelization are small. Figure 4.13 and 4.14 show the results for the homogeneous and heterogeneous experiments. Additional results are shown in Figures B.7 to B.12 in Appendix B. In the homogeneous case with a SPM restriction of 512 B the results show a usage of polling-, interrupt- and sleep-based version. We could also obverse solutions using combined FIFOs and mapping of up to one channel to the SPM. Raising the SPM restriction enabled to GA to map more communication onto this memory. In addition, the results show an increased usage of combined FIFOs. Furthermore, with more available SPM space the GA maps more task in data to this memory. We observe that almost all channels were mapped to the SPM and the majority uses combined channels for the SPM setting of 8 KB. We believe, that the outliers in the combined memory footprint (cf. Figure 4.13(a)) could be removed if more individuals where simulated.



(a) Run time and energy consumption



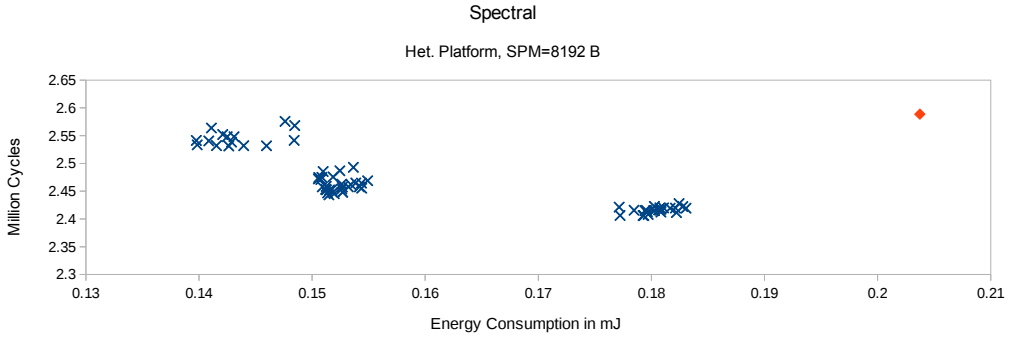
(b) Combined memory footprint and energy consumption

Figure 4.13: Spectral on the homogeneous system with SPM restricted to 2048 B.

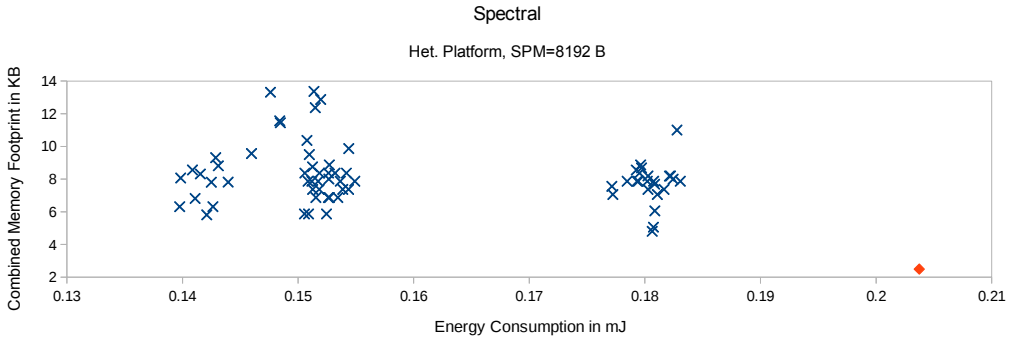
For the heterogeneous case, we observe similar FIFO types and SPM mapping for a restriction of up to 2 KB. Thus, increasing the number of generations should smoothen the results. In the 8 KB case, we observe three solution islands. The left island contains more energy efficient solutions and use mostly polling and sleep-based types for single channel FIFOs. Most of the channels are merged into polling and interrupt-based channels. The middle island has only a few solutions which uses sleep-based single channels, all other prefer combined channels. As for the left island, the merged channels use polling and interrupts. For the right island, we see that the ratio of polling-based FIFOs increases.

This observation is in line with the fact that polling is faster but energy inefficient. The `Filterbank` shows a similar behavior under some conditions.

The results for the combined memory footprint indicate that the `Spectral` benchmark does not improve much with larger channel sizes. This was expected due to the very tight loop-carried dependencies. However, PICO was able to generate a broad range of solutions which improve the performance in comparison to the baseline. In terms of energy consumption, the GA found solutions which consumed 31% less energy compared to the baseline solution. PICO found solutions with a run time improvement of 7%.



(a) Run time and energy consumption

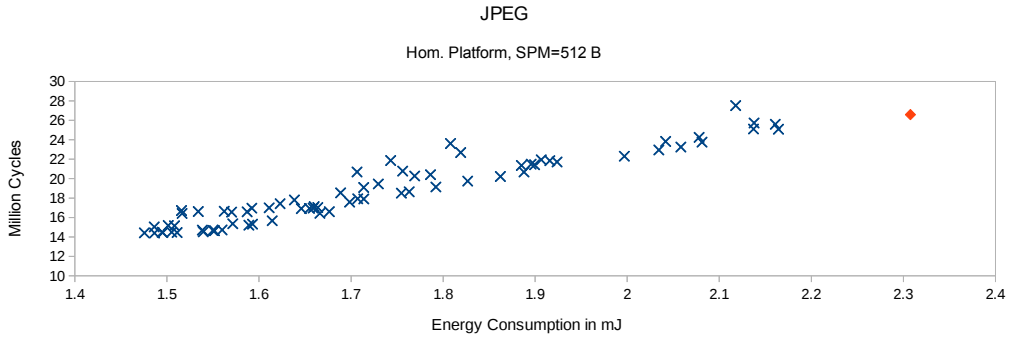


(b) Combined memory footprint and energy consumption

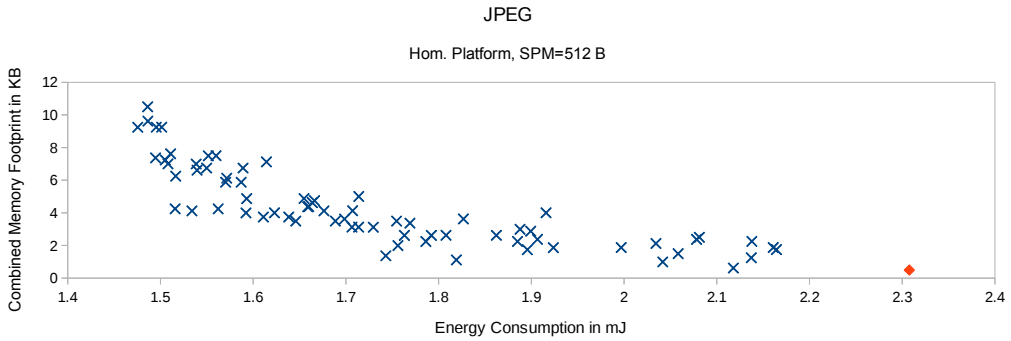
Figure 4.14: Spectral on the heterogeneous system with SPM restricted to 8 KB.

JPEG: The `Filterbank` and `Spectral` results did not clearly indicate that the channel capacity might have an influence of the overall performance. Figure 4.15 and 4.16 show the results for `JPEG` benchmark. Additional results are listed in Figures B.13 to B.16 in Appendix B. As the results show, already with a small SPM of 512 B, PICO could map data to this memory. In addition, the test case indicates a relation with run time and memory footprint. In contrast to `Spectral` and `Filterbank`, the `JPEG` application seems to profit from larger FIFO sizes. This can be observed in all analyzed SPM configurations and systems.

For the homogeneous case with a SPM of 512 B (cf. Figure 4.15), the results use polling-, interrupt- and sleep-based single channel data synchronization. Most of the solutions mapped one or two FIFOs to the SPM, rare cases used SPM or DRAM

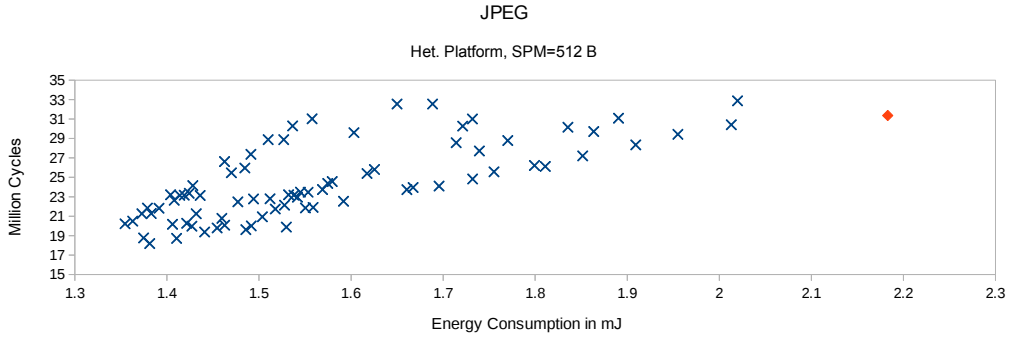


(a) Run time and energy consumption

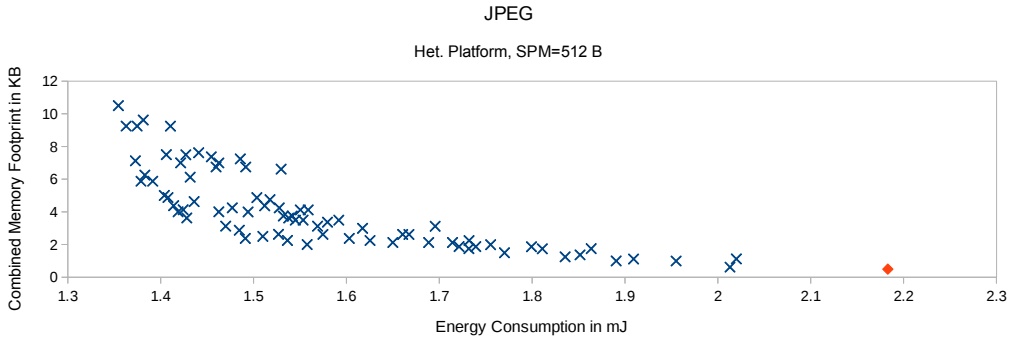


(b) Combined memory footprint and energy consumption

Figure 4.15: JPEG on the homogeneous system with SPM restricted to 512 B.

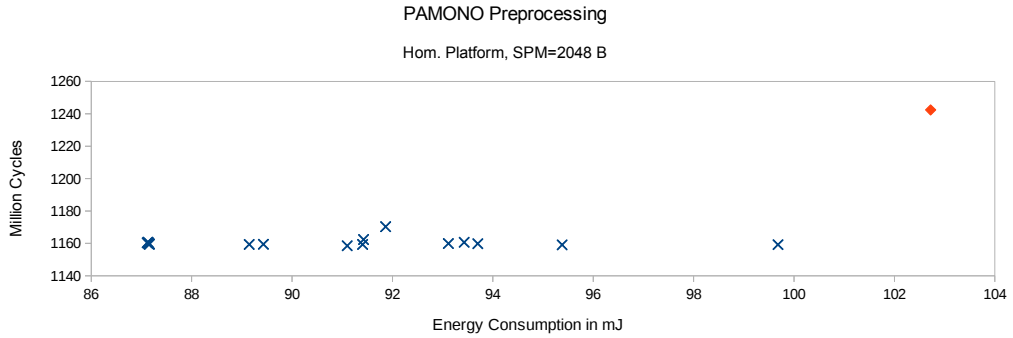


(a) Run time and energy consumption

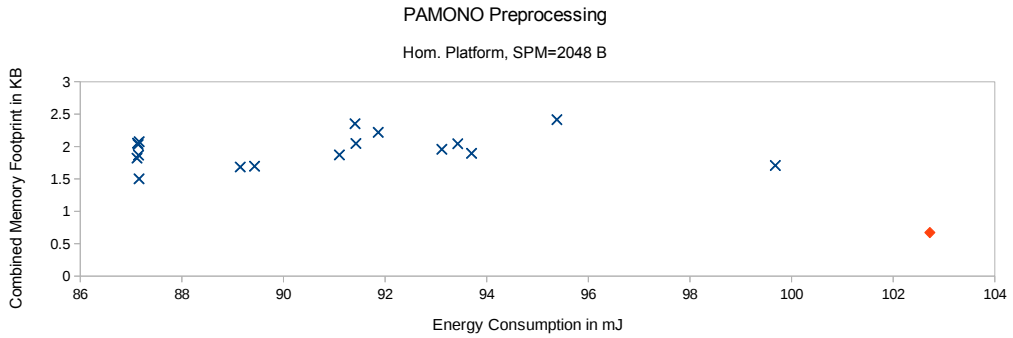


(b) Combined memory footprint and energy consumption

Figure 4.16: JPEG on the heterogeneous system with SPM restricted to 512 B.



(a) Run time and energy consumption



(b) Combined memory footprint and energy consumption

Figure 4.17: PAMONO preprocessing on hom. system with restricted SPM to 2048 B.

exclusively. The same mappings could be observed for the other settings. The results indicate that a SPM of more than 1 KB does not improve the performance significantly.

In general, PICO was able to generate a large variety of solutions if different characteristics and the user select the solution which fits the needs best. For the `JPEG` experiments PICO was able to find solutions which reduce the run time to 54% of the baseline which is a speedup of 1.84. In addition, for these solutions, the energy consumption could be reduced by 37%. But, these solutions have a larger memory footprint than the baseline.

PAMONO Preprocessing: The `PAMONO preprocessing` required by far the longest evaluation time per individual of all the benchmarks. A single simulation could easily reach the one-hour mark. We varied the SPM restriction between 512 B to 2048 B but observed similar results. Figure 4.17 shows the result for a SPM of 2048 B. Figure B.17 and Figure B.18 visualize the other results in Appendix B. Overall, the results show that there is some potential for improvements and PICO generated a broad solution front. It seems, that this benchmark profits from larger SPM sizes. This is especially visible in the FIFO type mapping. In the case of a SPM of 512 B, the solution set contains almost no RTEMS message queue-based synchronization. Here, the GA used the other single channel implementations. If the SPM is increased, the results comprises more RTEMS-based exchange. For all configurations the GA did not use a combined channel. The results offer solutions with an energy reduction of roughly 15% compared to the baseline implementation and in terms of execution time an improvement of 7%.

	Execution time			Energy consumption		
# Bytes	Model	Simulation	Diff	Model	Simulation	Diff
1	724	743	2.62 %	20.74	21.00	1.25 %
4	729	764	-1.93 %	21.88	21.62	-1.22 %
20	822	887	7.91 %	22.93	25.11	9.49 %
40	1024	1063	3.81 %	28.88	30.02	3.94 %
80	1466	1485	1.30 %	41.02	41.42	0.97 %
160	2748	2618	-4.73 %	72.17	68.42	-5.20 %
316	7566	7298	-3.54 %	214.98	206.20	-4.09 %
1024	29621	30055	1.47 %	848.76	867.28	2.18 %
4096	125234	127824	2.07 %	3592.50	3644.66	1.45 %
16384	523664	514397	-1.77 %	15724.51	15439.94	-1.81 %

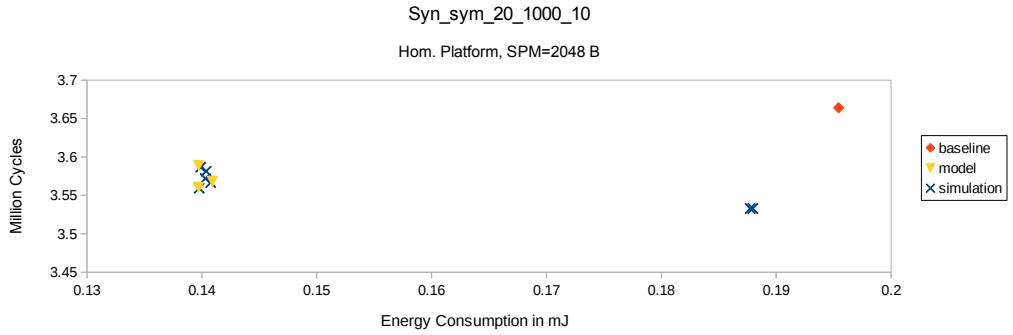
Table 4.3: Write to a polling-based FIFO with a capacity of 1 mapped to the SPM.

4.5.3 Model-based Optimization Results

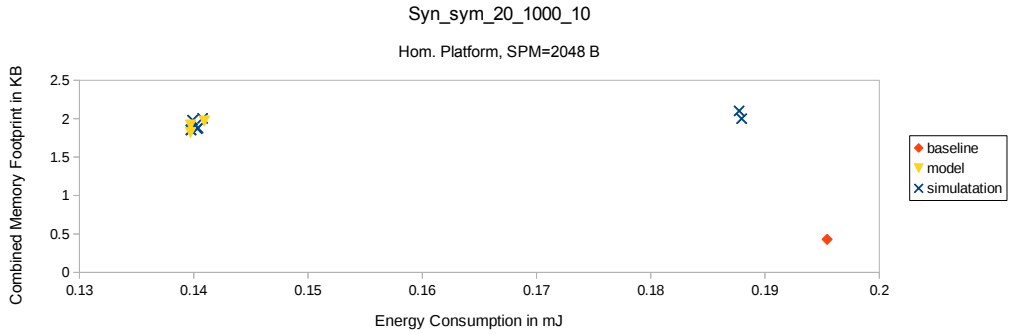
The run time of the optimization is dominated by the fitness evaluation. As the previous results showed, the optimization can take more then a day. Therefore, we proposed a high-level estimation (cf. Subsection 4.4.2) which should improve the overall run time. In the following, we present how we gathered the underlying data used by the model to estimate the performance. Finally, we present first results of a hybrid GA optimization run.

Model Evaluation: We created several simple test cases and measured the performance with different FIFO configurations to tune the model. Sleep calls were used to force a blocking FIFO access. The collected execution and energy consumption values were feed into a regression analysis. Table 4.3 shows the results of the model evaluation for a polling-based FIFO mapped to the SPM. The model fits well for this setting. However, in case of DRAM write access the values differed more. In the worst case, the difference was roughly 30%. This was mostly observed for smaller data. Section A.4 provides a comprehensive overview of the other results and the equations derived from the regression. We also observe mispredictions for the interrupt-based FIFO. This is obvious since predicting interrupts is hard. However, the provided FIFO implementations varied in their performance characteristics. For instance, a one byte write access to a RTEMS queue takes much longer compared to the polling-based implementations. But, writing larger data to the RTEMS is more efficient. This fact is captured in our model. Therefore, we expect that a hybrid GA optimization keeps the induced errors in an acceptable range. In such a case, a simulation run would from time to time correct these errors.

Figure 4.18 shows the results of a model-based and simulation-based evaluation for one of the synthetic test cases. We used the same settings for both experiments. To compare the results, we simulated the Pareto points from the model-based evaluation. The evaluation time for the model-based evaluation including the simulation of the



(a) Run time and energy consumption

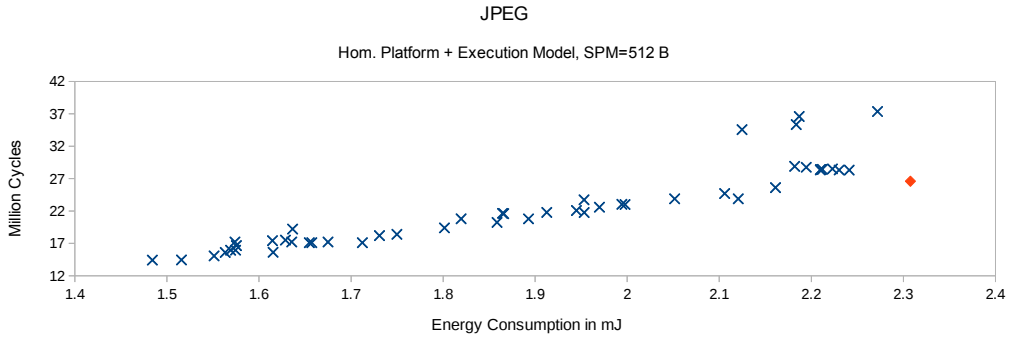


(b) Combined memory footprint and energy consumption

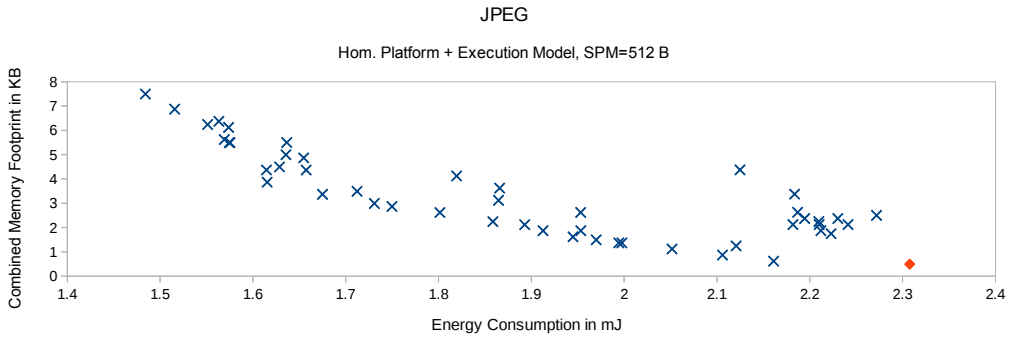
Figure 4.18: Model-based and simulation-based evaluation with SPM restricted to 2 KB.

Pareto-frontier was roughly 27 minutes and for the simulation-based 3 hours and 26 minutes. As the results indicate, the model finds similar solutions. The configurations on the right side are marginal faster and were not found by the model. We expect that this comes from the high-level abstraction of the model. Both experiments resulted in similar combined FIFO channels settings. Therefore, we conclude that the model-based evaluation evolved in the same direction as the simulation-based.

Hybrid GA Optimization: In this experiment, we configured the GA to use for up to 50% of the generations the execution model-based fitness evaluation. In addition, we ensured that for the first and last generation the simulation-based fitness evaluation is used. The high-level execution model, at the current state, only calculates the performance for the parallel regions. Therefore, to compare results obtained by the model-based with the simulation-based we need to re-evaluate the model-based results first. In the case that a simulation-based generation encounters a stored fitness for a model-based individual, the individual gets re-evaluated on the simulator. Further, we configured the GA in such a way that at least two succeeding generations were evaluated with the execution model. This should ensures that not all individuals get re-evaluated. Finally, we model the distance since the last simulation-based generation, if the distance increases, it is more likely that the new generation will be evaluated with the simulator. For this experiment, we configured the system in such a way, that at least two and at most five succeeding generation the model-based evaluation use. This should keep the estimation



(a) Run time and energy consumption



(b) Combined memory footprint and energy consumption

Figure 4.19: JPEG with SPM restricted to 512 B and hybrid GA.

error introduced due to the high-level model and inaccuracy especially for interrupt-based synchronization in acceptable ranges.

For this evaluation we selected the `JPEG` test case on the homogeneous platform with an SPM restriction of 512 B. The pure simulation-based optimization took more than 20 hours whereas the hybrid optimization took roughly 10 hours and 40 minutes. The algorithm performed 45,798 genetic operations which is similar to the simulation-based case. In total, the GA evaluated 2098 individuals where 1142 were simulated and the remaining fitness values were derived from the execution model. Figure 4.19 shows the results of the hybrid evaluation the solutions are similar to the simulation-based only approach shown in Figure 4.15.

4.5.4 Discussion

The evaluation highlights the importance of a holistic view on data synchronization with respect to implementation details, mapping and channel capacity. It must be emphasized again, that this evaluation focuses on the capabilities of PICO and not on the prototype FIFO implementations which are considered as a black box. Therefore, the resulting execution time, energy consumption and memory footprint heavily depend on the target platform and prototype implementation. For all experiments, the structure of the resolutions indicate that PICO was able to explore the solution space and find solutions outperforming the baseline in some objectives. For instance, our optimization

algorithm found in the `JPEG` experiment solutions with a speedup of 1.84 or a reduction in energy consumption by 37%. Even in cases which suffer from tight data dependencies, PICO was able to offer solutions which were more energy efficient. In some scenarios, a reduction of energy consumption lead to a longer execution time due to the utilization of sleep states. However, in (soft) real-time this might be acceptable if the application still meets the deadline.

A comprehensive optimization run takes time, therefore we used a set of synthetic benchmarks to obtain general rules applicable for fast prototyping. Overall we can say that the performance of an application usually benefits from larger FIFO capacities. Ideally, the most used channels should be mapped to the fast and energy efficient SPM. Merging of channels can be beneficial if the time span between the data, mapped to the combined FIFO, is written is rather short. The same must be true for reading, otherwise the execution might be unnecessary delayed. In the case of unbalanced parallelism, a FIFO implementation which utilizes sleep states might be beneficial.

The detailed measurements to tune the execution model revealed another aspect. Depending on the element size, we observed a big performance bottleneck for the target platform if the data is not aligned properly in the memory. This heavily depends on the target architecture's memory layout, hierarchy and how the access is realized. Therefore, it might be beneficial to either add padding data or adjust the FIFO implementation to respect aligned memory access.

The hybrid model-based optimization produced promising results. This approach was able to reduce the overall optimization time significantly but was still able to find well performing solutions. Overall, a holistic model-based approach which does not require a simulation seems to be a worthwhile goal. Therefore, we need to further tune the model to improve the estimations. That may be an application for machine learning.

This evaluation showed that the static fitness evaluation of the GA works. The algorithm sorted out individuals which exceed the memory restriction before a time-consuming simulation. In addition, PICO's result data base prevented meaningless re-evaluations of known configurations. In such a case, the GA performs additional genetic operations to create new unknown individuals. In the current implementation, PICO performs up to 10 operations. As the results indicate, especially in the case of tight memory bounds this might not be sufficient. Therefore, we propose three strategies to deal with this issue. Firstly, the number of genetic operations performed could easily be increased. Secondly, a larger number of generations would increase the probability for new unknown individuals. Finally, an enhanced repair function might be able to modify problematic individuals in such a way that they are valid and unknown. Therefore, the repair function needs more context and structural knowledge about the target platform and capabilities of the provided channel types. Then it might be possible to correct the genes causing the issue. For instance, in the case of exceeding the memory restriction, a better suiting capacity might be calculated.

In this evaluation, the fitness evaluation as well as the Pareto dominance function

considered memory consumption for both memories. PICO can be configured such that it only considers execution time and energy consumption. This setting can be used to explore the general nature of the target application.

The evaluation revealed the importance of a multi-objective aware communication optimization for the PA4RES framework. The broad range of solutions and their improvements must be considered during the parallelization. Currently, PAXES only accounts with a fixed cost for data synchronization similar to the baseline used in the experiments. A feedback loop, which provides the results found by PICO back to PAXES seems a worthwhile extension. Again, a possible application for machine learning.

4.6 Conclusion and Future work

Data synchronization in parallel applications is a vital performance bottleneck. An efficient data exchange is crucial, and if done improperly, wastes potential benefits gained through the parallelization. PICO provides a holistic approach to provision the data communication automatically for a given application towards a target platform with respect to the objectives execution time, energy and memory consumption. Using an evolutionary algorithm, PICO explores the capabilities of the target hardware and software stack efficiently. In this chapter, multiple exemplary FIFO implementations were utilized to demonstrate the capabilities and benefits of PICO's communication optimization approach. The fitness evaluation can either use the (simulated) target platform to measure the performance or consult a high-level execution model. In general, this approach considers the offered FIFO implementations as black boxes and thus can be applied for other systems.

We demonstrated the capabilities of PICO's communication optimization algorithm with several real world and synthetic test cases. The GA was able to generate a broad range of solutions with different performance characteristics. For instance, PICO found solutions with a speedup of 1.8 or a reduction in energy consumption of 30% compared to a baseline implementation. In other cases was the GA able to provide more energy efficient candidates. These results highlight the importance of the actual realization of the data synchronization. Therefore, we conclude that the simple data exchange mode used by PAXES hides potential promising solutions. The execution model offered a promising reduction in overall optimization time. However, the evaluation also revealed that the underlying data that the model uses could be improved.

The evaluation proves that the questions raised in the introduction have an impact on the overall execution. Especially, the results show that there is no general rule and some solutions perform better for some applications than others. Fortunately, the novel communication optimization approach developed in this thesis supports the developer. PICO helps to find answers regarding memory mapping, channel size, implementation details and channel merging.

Additional improvements could be observed with an extension to the optimization algorithm which moves the communication nodes in the graph. Further, it might be worthwhile to implement the windowed FIFO [HSH09] semantics into PICO. A more precise execution model would improve the performance estimations especially modeling of contention seems promising. Therefore, we expect that adding more data points to the regression analysis would lead to an improvement. Overall, a holistic model-based approach which does not require a simulation seems to be a worthwhile goal.

The GA generates new unknown solutions and evaluates them with a costly fitness evaluation. However, if such a solution is worthwhile to simulate is currently unknown and thus (costly) evaluated. Fortunately, model-based optimization (MBO) offers a method to predict if an individual solution should be evaluated to gain new knowledge or not. We used MBO in the context of the CRC and in [KLN17] to estimate run time and performance of specific machine learning algorithms. Therefore, to prune the evaluation time of PICO's genetic algorithm, a combination with a MBO approach seems promising.

Currently the entire PA4RES parallelization process is split into PAXES and PICO. PAXES assumes a static FIFO implementation and only uses a fairly abstract high-level communication cost model. Knowledge discovered by PICO regarding the actual synchronization impacts could be fed back into PAXES' parallelization algorithm to further improve the results. Finally, we would like to analyze PICO's capabilities regarding other benchmarks and new hardware platforms.

Chapter 5

Emerging Challenges for Embedded Systems - Real-time Virus Detection

Contents

5.1 Introduction

Today's growing demand for computation power and energy efficiency to realize new complex mobile applications using high-performance embedded systems require sophisticated solutions. Such applications often stem from the domain of computer vision solving recognition tasks or machine learning. Exemplary applications are mobile robotics applications or collision avoidance and autonomous driving systems in modern cars. These

scenarios expose new challenges to high-performance embedded systems. Despite several hardware platform parameters, the software might offer huge parameter sets to configure the application behavior. Determining the performance impact of these parameters is a complex and time-consuming task. This chapter presents our (automatic) exploration approach to find reasonable solutions in a large parameter space. With a software-based biological virus detection, we demonstrate the capabilities of our Design Space Exploration (DSE) algorithm to explore software and hardware parameters efficiently. Previously this detection algorithm was executed on desktop computers only. In our experiments, we achieved soft real-time capable solutions running on a high-performance embedded system demonstrating the feasibility of a mobile battery-driven application.

In recent years, modern high-performance embedded systems such as the Odroid-XU3 (cf. Subsection 2.1.2) offer improved performance and capabilities. These heterogeneous systems provide different characteristics combined into a single MPSoC. In case of the Odroid-XU3, the embedded Mali-T628 GPU offers enough computational power to enable the usage of this platform for the computer vision domain. However, the large parameter space offered by these heterogeneous multi-core systems exposes a significant challenge to embedded system developers in order to find good parameters to obtain the necessary performance with respect to available energy budgets or peak power consumptions. Despite the platform parameters, the software which should be executed on the platform is highly configurable and thus increases the complexity of DSE. Finding good solutions in a huge parameter space is a complex exploration task. Based on an existing DSE algorithm [LMS14] we developed a hardware-in-the-loop, multi-objective aware DSE specialized for embedded systems. Section 5.3 presents more details on the DSE algorithm. In this thesis, we use a biological virus detection as a driving application.

Today, in our connected world where we can reach remote areas within hours by plane, viruses and thus diseases can spread easily over the whole planet in a short period of time. Therefore, a fast and reliable virus detection is crucial to contain the spread of viruses. Devices which enable such virus detection are more and more demanded. To analyze samples in remote areas, a mobile solution is desirable.

Most available computer vision-based approaches require complex computations. Consequently, these systems are large, heavy and immobile. For this reason, they are typically bound to specific locations like laboratories or hospitals. A portable solution is preferable since it would drastically increase the versatility of such detection systems and help to contain virus-caused diseases in remote locations. The Plasmon-Assisted Microscopy of Nano-Objects (PAMONO) sensor [Za10; SSL17; LSS17] and its software processing pipeline in combination with the Odroid-XU3 are a promising candidate for a mobile virus detection solution. This biosensor can detect and count individual viruses in liquid samples within less than three minutes. Section 5.2 gives details on the sensor and Section 5.4 discusses the detection software.

For applications from the computer vision domain, adjusting the detection quality may be possible without loss of expressiveness. Changing the detection quality may

improve the run time and energy consumption and thus offers large optimization potential especially for constrained embedded systems. Chapter 6 discusses these optimization approaches which are commonly not considered in the embedded domain and so waste this opportunity. Considering this trade-off enables an additional optimization dimension for our DSE approach.

To summarize, this chapter presents our enhanced DSE framework which assists in finding good hardware and software configurations for a given application. We use a computer vision-based biological virus detection software using the PAMONO sensor as a use case. The DSE should find hardware and software configurations for the PAMONO application with low energy consumption, sufficiently fast evaluation speed, and acceptable virus detection quality. Especially, we are interested what hardware or software parameters alone contribute to the overall performance and if a combined hardware/-software codesign achieves better results. Furthermore, what is the improvement if we accept a degradation in the detection quality?

This chapter is based on work published in [NLE15] and [MFN17]. The author of this thesis had the initial goal to use an embedded system as a target platform for the virus detection. However, this work was a collaboration between Pascal Libuschewski and the author of this thesis. Additional information can be found in [Lib17]. Both authors contributed equally to this work. Together, we prove that high-performance embedded systems, combined with powerful detection algorithms, are a promising platform to enable battery-driven mobile virus detection.

This chapter is structured as follows. Section 5.2 provides the fundamentals of the PAMONO sensor. Section 5.3 gives a high-level description of the DSE and provides details on the extensions we made to support embedded systems. Section 5.4 presents the virus detection software and the parameters that the DSE can explore. Section 5.5 presents the impressive evaluation results. Section 5.6 presents related work and Section 5.7 concludes this chapter and gives perspectives for future work.

5.2 Plasmon-Assisted Microscopy of Nano-Objects

Classic optical methods fail for the detection of nanometer scale physical structures, like viruses. Light-based microscopes are limited by the available wave length and the numerical aperture of the device. The Abbe diffraction limit [Abb73] describes this fundamental limit. Theoretically, the observation of viruses is possible but requires exceptional good and thus expensive microscopes. Therefore, such structures are traditionally observed using indirect approaches. Most methods require a certain incubation time and reaction of the patients immune system and thus increase the time span to a sufficient treatment. For instances, the hemagglutination assay of a blood sample can be used to attest the antibodies of a specific virus. Depending on the virus type, these tests usually require stationary laboratories and specially trained personal.

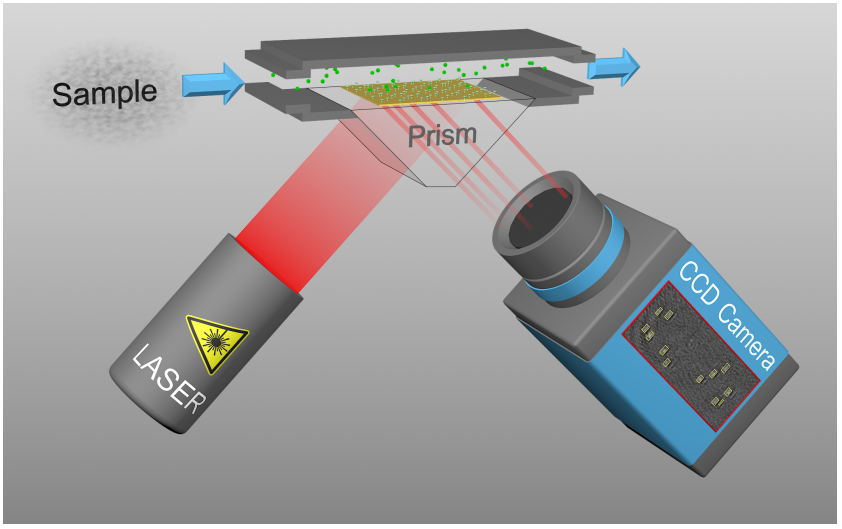


Figure 5.1: PAMONO sensor overview [NLE15].

The "Leibniz-Institut für Analytische Wissenschaften - ISAS - e.V." [Lei18] developed the Plasmon-Assisted Microscopy of Nano-Objects (PAMONO) sensor which is able to analyze samples for small particles using indirect optical methods [Za10; SSL17; LSS17]. We have access to a prototype sensor, samples and results through the collaborative research center 876. In the following, we provide a high-level description of the sensor, more details can be found in the related publications.

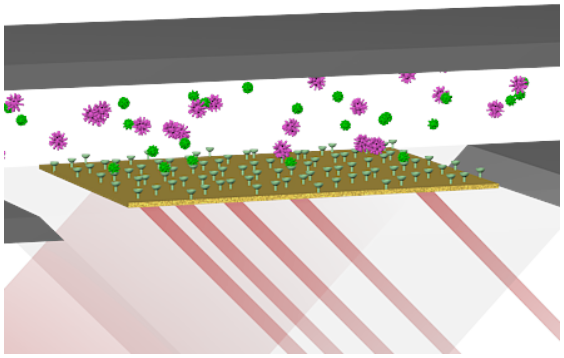


Figure 5.2: Virus adhesion process visualization [NLE15].

Figure 5.1 shows the overall layout and the important components of the PAMONO sensor. The flow cell is the main part of the sensor system (top most). Inside this cell is a thin gold layer coated with antibodies. During the analysis phase, the sample is continuously flowing through this cell and viruses will attach onto the antibodies. After the adhesion process, attached viruses remain on the gold layer through the analysis cycle. This implies, that the current sensor is only sensible to a specific virus stem which reacts with the antibodies and is insensible for other stems. The second component of the sensor is a laser illuminating the bottom side of the gold layer through a prism. The reflected light of the gold layer is then recorded by a CCD camera. The laser light excites so called plasmon waves within the gold layer.

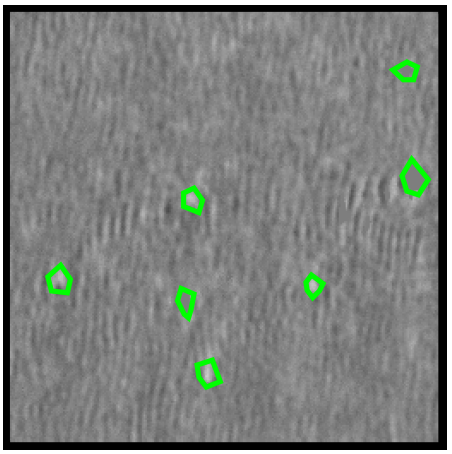


Figure 5.3: Virus adhesion process visualization [NLE15].

The excitement of the plasmon waves highly depends on the layer thickness. Thus, small changes in the layer's height strongly affect the plasmon waves and accordingly the strength of the reflected light. Such effect is observable when a virus attaches to the antibody layer. Therefore, even "if a virus is *invisible* for the light, the attaching process can be observed on a micrometer scale as a small, faint spot appearing in the images recorded by the camera" [NLE15]. Figure 5.2 shows a more detailed view of the flow cell with the gold layer as well as attached viruses and the changes in the reflectivity of the layer. Figure 5.3 shows a recorded image, for visualization purposes we marked attached particles. As the figure shows, the recorded image is very noise and the detection of viruses is hard, especially if dirt particles additionally distort the image.

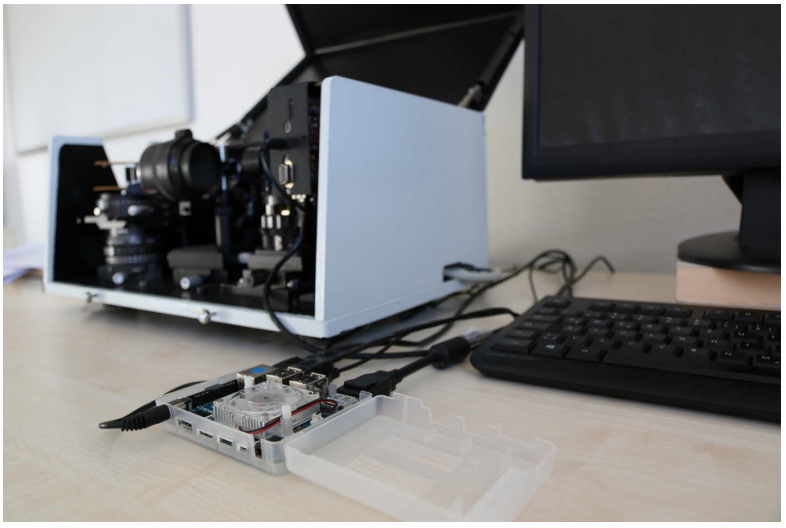


Figure 5.4: Odroid-XU3 (front) and the PAMONO prototype sensor (back).

Figure 5.4 shows the Odroid-XU3 in front and the prototype PAMONO sensor setup in the back. The prototype sensor currently uses exchangeable (large) components like an actual camera lens. We think that the sensor itself could be drastically shrunk in size if necessary, e.g., for a real world mobile usage.

The PAMONO sensor detects and counts the viruses and does not attest antibodies as in contrast to rapid tests for virus infections like the flu or HIV. Therefore, the PAMONO sensor can detect the viruses as soon as they show up in the sample and therefore it closes the gap between the time the patient is becoming contagious and the patient's immune system is developing the first antibodies [NLE15]. Originally, the collected PAMONO sensor data was analyzed manually in an extensive time-consuming process. Fortunately, in the context of the Collaborative Research Center (CRC) 876 Siedhoff [Sie16] and Libuschewski [Lib17] developed an automatic computer vision-based detection software reducing the analysis time significantly. Section 5.4 gives more detail on the software. However, such complex software offers versatile parameters to tune the application towards the requirements, Section 5.3 presents our automated approach to tackle the large parameter space.

5.3 Design Space Exploration Framework

Developers often face the task to configure a complex application and a manifold target platform to meet certain requirements like run time or energy budget. Complex software and modern hardware platforms offer a wide variety of parameters. Some parameters are related to each other and thus changing one parameter might conflict with other parameters resulting in bad performance. Considering all relations and side effects is complex and thus an automated holistic approach is necessary to support system developers. This chapter presents the Design Space Exploration (DSE) framework which automatically identifies hardware and software parametrization. The DSE is evaluated with several other applications from a wide variety of domains, although we focus on the PAMONO sensor application. In this chapter we focus on the necessary extensions to the DSE to enable embedded systems as a target platform. More details on the DSE can be found in [Lib17]. However, in the following we provide a brief introduction to our DSE framework.

Genetic Algorithms (GAs) are often used for parameter exploration especially to explore a highly non-linear design space. For instance, PICO uses a GA to improve the data exchange between concurrently running tasks (cf. Subsection 4.4.1). The DSE presented in this chapter is based on a heavily modified Java-based Evolutionary Computation Research System (ECJ) [LPB18]. We use NSGA-II [DAP00] for the multi-objective evaluation provided by ECJ. Previous iterations of the DSE framework targeted only a GPU (simulator) for the fitness evaluation. However, in this study we use a complete real (embedded) hardware platform, so we adapted the framework to support arbitrary target platforms. We generalized the DSE. Hence, it is now able to accept fitness evaluation results from various sources. To evaluate the energy consumption and execution time of the application executed on the Odroid-XU3 system, the DSE interfaces with the EnergyMeter tool (cf. Subsection A.2.1). For this reason, the DSE framework now considers the energy consumption of the GPU, processors and memory

whereas the previous iteration only supported (simulated) GPUs. We enhanced the DSE to explore additional hardware parameters like processor clock frequencies or applied governors. Further, we extended the fitness evaluation capabilities into the direction of approximate computing. It is now possible, that solutions with a degraded result quality survive due to their better performance in terms of energy consumption or execution time.

As previously mentioned, parameters can be in a relation to each other or must be within certain value ranges. For example, if one parameter is set to a specific value, the related parameter may only use a restricted value range. The DSE framework allows the user to model such parameter relations as well as value ranges and restrictions with an input specification file. For instance, the core frequencies of the Odroid-XU3 can only be changed in 100 MHz steps.

During the optimization process, the DSE runs on a master computer to manage the exploration. The target system is connected e.g. via network or software socket to the master. If available, multiple target systems can be utilized to speedup the evaluation phase. The DSE distributes the individual solution candidates including all necessary files automatically to the target and starts the evaluation. To increase the accuracy of the evaluation results on a real (hardware) system, the experiment can be repeated several times automatically. Finally, the (averaged) fitness values are collected and reported back to the master system. Once a generation is evaluated, the GA generates a checkpoint containing all partial results and creates a new generation for the evaluation. This process is repeated until the termination criterion is met.

5.4 Use Case: Automatic Virus Detection Software

The PAMONO biosensor satisfies all requirements for a fast real-time virus detection. However, up till recently, the analysis results were evaluated manually which might take days. Recently, researchers [Sie16; Lib17] developed an automatic virus detection software for this promising sensor. The VirusDetection with OpenCL (VirusDetectionCL) software is capable to detect and count viruses in images recorded by the sensor's camera. As a computer vision application, VirusDetectionCL has huge performance demands onto the target platform. We think that such applications are emerging into the domain of mobile embedded systems and are thus raising interesting research questions. The combination of sensor, software and hardware is a complex Cyber-Physical System (CPS). As Section 5.5 shows, this application is a good test cast for the DSE. The DSE is able to explore and solve problems exposed by emerging applications in the (mobile) embedded systems domain.

During the analysis phase, the system must process several hundred images to obtain a reliable result in a short period of time. In addition, it is desired, but not required, to process the images, recorded by camera in *real-time*. The used camera records images with 25 frames per second. By meeting this soft real-time requirement, image buffers

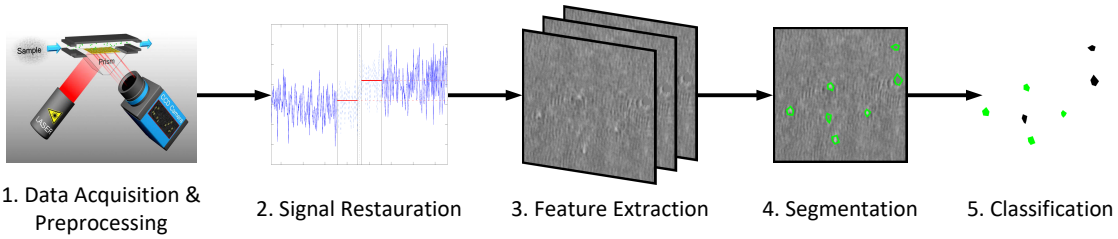


Figure 5.5: Virus detection pipeline overview, based on [NLE15].

between the camera and in the succeeding detection software can be removed or at least scaled-down. Further, this enables a theoretically infinite execution without the risk of buffer overflows. If buffers are still required, smaller buffer and thus smaller memory sizes are still beneficial. Since memory is expensive especially for embedded systems, this might also reduce the cost of the system.

Figure 5.5 visualizes the conceptional work flow of the detection algorithm. The software works as a five stage pipeline. The first stage acquires the images from the sensor. The pre-processing step uploads 16-bit gray-scale images to the GPU and converts them to floating point arrays. The signal restoration step uses the physical signal model of the PAMONO sensor to restore the signal of the attaching virus particles. Further, this step removes noise and the constant background signal. Some of the processing works on series of images. These two presented stages provide the fundamentals for one of the benchmarks used in the PICO evaluation (cf. Section 3.7). During the feature extraction phase, different per-pixel and per-polygon features are calculated. The per-pixel features can be interpreted as a pixel's degree of membership to a class of pixels corresponding to a virus adhesion. The per-polygon features represent the degree of membership of the whole polygon to the class of a virus adhesion. The extracted features are then segmented by applying the polygon structures around prominent areas. Finally, the classification step decides if the polygons correspond to viruses or not. In the ideal case, each virus adhesion in the images is identified by a corresponding polygon and the polygon size should match the size of the virus adhesion.

The version of the virus detection software used in this thesis is written in C for the Central Processing Unit (CPU) code and OpenCL for the GPU code. It consists of $14,000$ SLOC of C code and $4,000$ SLOC of OpenCL code. The virus detection software also includes OpenCL host code and GStreamer [GSt18] dependencies. The dependency to OpenCL and thus special hardware support limited the detection software to desktop computers. However, modern mobile systems like the Odroid-XU3 provide sufficient OpenCL support to execute the detection pipeline. This section highlights the parameters of the hardware and software the DSE can explore to optimize the systems' performance.

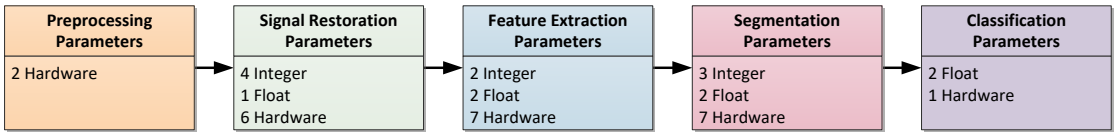


Figure 5.6: Pipeline parameter overview, based on [NLE15].

5.4.1 Implementation and Parameter Details

Figure 5.6 illustrates the detection pipeline and the number of configurable parameters for each pipeline stage. Subsection 5.4.1.1 provides more details on the configurable hardware-related parameters. The parameters for the signal restoration phase mainly influence the noise reduction quality. The virus detection software provides several feature extraction algorithms. Here, the pipeline parameters determine detection thresholds and the selection of feature extraction algorithms. The segmentation parameters, mainly thresholds, influence how the polygons are created and how the extracted features per pixel are combined to features per polygon. Finally, the classification parameters sort false detections for the set of polygons out. To conclude, the detection pipeline provides 23 hardware and 16 software parameters. In the following we present a more detailed description of the hardware parameters since they are in the focus of this thesis. [Lib17] provides more details regarding the software parameters.

5.4.1.1 Hardware-related Parameters

The virus detection software provides various hardware-related parameters which can be used to tailor the application towards a specific target hardware. These parameters mainly focus on OpenCL configurations and thus relate to the GPU. Major parts of the image processing is implemented in OpenCL and thus effects from these hardware-related parameters can influence the overall performance heavily. The work group size controls the parallelism in that way that it defines the number of concurrently running threads allocated onto the GPU. For instance, if the work group size for the noise reduction is set to 16×16, $16 \times 16 = 256$ threads for the calculation. These threads are then scheduled on the GPU, in case that the GPU can not handle all threads simultaneously they got executed in multiple steps. Data synchronization between concurrently running threads is essential, in this case, partial results within the work group are shared through the shared memory on the GPU itself. In case of the Odroid-XU3, the GPU and processors use the same memory. The memory partition may influence the detection quality. In these cases, the algorithm uses knowledge provided by previously processed images for the calculation, e.g. during noise reduction or feature extraction. For this reason, if more images are available, the algorithm usually generates more precise results. Therefore, depending on the memory allocation the detection quality of the virus detection algorithm varies. More details regarding these hardware-related parameters in the virus detection software can be found in [Lib17].

Despite the hardware-related parameters provided by the algorithm, the Odroid-XU3 platform also provides some configurable hardware parameters. Most important is the ability to configure the clock frequencies of the A15 and A7 processors. These hardware parameters are important for code executed on these processors. The following section provides more details for this platform feature.

5.4.1.2 Dynamic Frequency Scaling

The processors used on the Odroid-XU3 platform can be clocked with configurable frequencies. Changing the frequency drastically influences the power requirements of the processor. Thus, Dynamic Frequency Scaling (DFS) is a key technique to reduce the energy consumption of the system. Fortunately, the Odroid-XU3 platform provides DFS accessible through the Linux operating system. In general, two strategies are available, one is running the core at a fixed frequency. The other changes the frequency dynamically depending on configurable switching policies. The system provides several governors that control the behavior of the processor. Further, we added additional standard governors which were previously not available in the Linux distribution shipped with the Odroid-XU3. The static governors are rather simple and do not add a large additional overhead. The *userspace* governor allows a manual specification of the frequency. Using the *powersave* governor, the processor runs at the lowest possible frequency. In contrast, the *performance* governor sets the processor to the highest frequency.

On the other side, the dynamic governors adjust the frequency during the run time considering the current workload of the processor. The *ondemand* governor increases the frequency if the average work load on the processor exceeds a given threshold and decreases speed if the average workload drops below that threshold. The governor instantaneously changes the frequencies and thus should not reduce the overall performance of the system drastically. The user can configure the governor in terms of sampling rate, threshold, down-sampling factors, and the power save bias. Frequencies are changed more aggressively toward the maximum clock rate by the *interactive* governor. This governor reacts faster to increasing work loads. Finally, the *conservative* governor changes frequencies in smaller steps to avoid peaks in power consumption. The user can define the step size, default is 5% of the maximum frequency. In addition, the down-sampling and threshold are configurable.

On typical desktop systems the *ondemand* governor is the default setting. However, in the virus detection scenario, a different governor might increase the overall performance in terms of run time and energy consumption. For instance, a governor which matches the frequencies according to the rate in which new images are captured might improve the overall performance. Finding a good solution manually is quite tedious, so we created a small tool which makes these governors accessible by the DSE.

5.4.2 Detection Quality

Pattern recognition applications are traditionally assessed regarding their detection quality. In this work we used the F_1 score (also F-measure), which is defined as the harmonic mean of precision p and recall r:

$$F_1 := 2\frac{p \cdot r}{p + r} \tag{5.1}$$

where the precision p is defined as:

$$p := \frac{TP}{TP + FP} \tag{5.2}$$

and the recall r is defined as:

$$r := \frac{TP}{TP + FN}. \tag{5.3}$$

Correctly detected viruses are called true positives TP. False positives FP are particles which are falsely classified as viruses and missed viruses are false negative FN.

Following these equations, the detection quality indicates how accurately the detection program counts the number of viruses in the sensor images. For example, an F_1 score of 1 says, that all individual virus were found in the images and no dirt particle or distortion has been falsely classified as a virus. Classic virus detection methods usually only attest a positive or negative contamination of a sample and do not indicate how strong it is.

Previously, the virus detection software was trained to achieve the highest F_1 score. However, in cases where a reduced detection quality is acceptable, new optimization opportunities arise. A degraded detection quality is applicable in cases where one is only interested if the analyzed sample is contaminated or not, in such a case, a detection quality of $F_1 \geq 0.5$ might be sufficient. Obviously, the acceptable F_1 thresholds heavily depends on the context in which the application is used. The detection quality can also be seen as the QoS or Quality of Results (QoR) of the complete system. Accepting a degradation of QoR is also the key technique of approximate computing (cf. Chapter 6). The evaluation (cf. Section 5.5) shows that variations in the detection quality can be traded against energy consumption and execution time.

The virus detection software offers several parameters which influence the QoR. For instance, the quality of the noise reduction or classification influences the overall QoR. Furthermore, the application may use input perforation or partial frame processing. However, the general prediction of the influence of parameters onto the detection quality, execution time and energy consumption is a complex task. Therefore, the improved DSE approach integrates these considerations.

5.5 Evaluation

The evaluation answers two questions. First, is the DSE able to explore the parameter space of the virus detection software for an real embedded platform efficiently? Second,

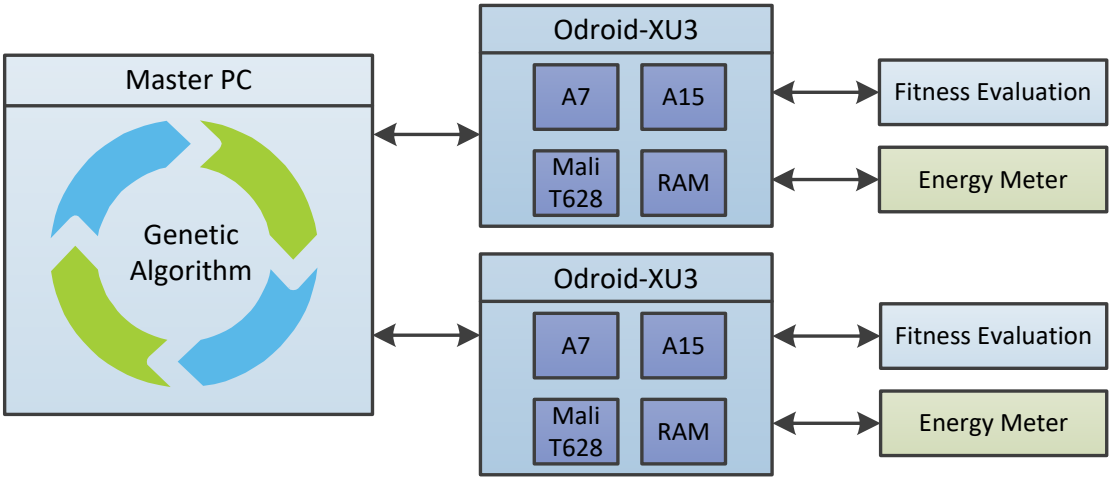


Figure 5.7: Evolution process: a master computer controls exploration and schedules individuals to the target systems for a parallel evaluation [NLE15].

since the virus detection requires a quite powerful system, can the DSE framework find promising configurations for a mobile (battery-driven) virus detection system? We conducted several experiments (cf. Subsection 5.5.2) on the Odroid-XU3 (cf. Subsection 5.5.1) to answer these questions. The promising results (cf. Subsection 5.5.3) indicate that a mobile real-time virus detection solution is already achievable with current technology (cf. Subsection 5.5.4). Notably, this was achieved without sacrificing detection quality.

5.5.1 Evaluation Setup

The Odroid-XU3 represents a modern powerful embedded platform. Subsection 2.1.2 provides more details of the system. For this evaluation we used two Odroid-XU3 boards and one master computer. Figure 5.7 visualizes the evaluation process. The DSE and GA run on the master computer and interact with the target systems. The experiments were conducted in a temperature controlled room and the temperatures were logged to exclude temperature-related effects. For example, in hot environments, the cooling fan of the board might be unable to cool the system and the system clocks down to prevent damage. The measured room temperature values were in a range of $22°C \pm 1°C$ and therefore we can exclude the influence of the ambient temperature. In addition, we configured the cooling fan control to run at maximum speed. To minimize the influence of other tasks running on the Odroid-XU3, we terminated all unnecessary services and applications before the evaluation.

Table 5.1 lists the configuration of the INA231 sensors used in the experiments. Regarding the configuration, the sensor measures shunt and bus voltages in continuous mode leading to continuous measurements instead of event triggered ones. The sensor averages over 16 samples for both shunt and bus measurements. Each single measurement converges over 4.156 ms resulting in an update period of 132.992 ms. Considering the expected application execution time, we expect that this measurement settings are sufficient to capture the application performance characteristics.

Parameter	Configuration
Number Of Averages	16
Bus Voltage Conversion Time	4.156 ms
Shunt Voltage Conversion Time	4.156 ms
Operation Mode	continuous
Update Period	132.992 ms

Table 5.1: INA231 [Tex13] configuration.

As described in Subsection A.2.1, the polling-based measurement may influence the fitness calculation, thus we assigned the EnergyMeter to the Cortex-A7 and excluded the processor from the DSE exploration and eventually from the final results. The program that should be measured runs therefore on the Cortex-A15 cores and the Mali-T628. However, the mapping of the application and EnergyMeter to the processors can be exchanged if desired.

As the analyzed program may produce some initial (large) overhead, e.g. to build the OpenCL kernels and fill all the queues, we setup the experiments in such a way, that only the main computation part is measured. This excludes the (static) initialization phase of the virus detection software which might otherwise hide the true program performance if the execution time per frame during the evaluation (e.g. number of processed images) is not long enough. Using named pipes, the algorithm interacts with the EnergyMeter and starts and stops the measurements accordingly. Therefore, only the real execution time of the detection algorithm is returned to the master computer.

5.5.2 Experiments

To explore the capabilities of the DSE and its applicability for our goal of a mobile virus detection system, we conducted three experiments. With Exp1_{hw} we restricted the DSE to explore only the hardware-related parameters provided by the Odroid-XU3 platform and the virus software. Subsection 5.4.1.1 provides additional information regarding these hardware-related parameters. With Exp2_{sw} we evaluated the potential solution space for software parameters only. Subsection 5.4.1 and especially [Lib17] provide more details regarding the software parameters configurable for the virus detection. In the final experiment $\text{Exp3}_{\text{hw\&sw}}$, we were interested if the DSE is able to explore a combined hardware/software codesign approach to gain additional improvements.

For all experiments, we used two data sets with 1,000 16-bit gray scale sensor images for the training and testing phase. Each image has a resolution of 706×167 pixels. The images are labeled, thus the positions and corresponding polygons of the appearing viruses within the images are known. The virus detection program generates a result file containing positions and corresponding polygons for the found viruses. With the labeled images and detection result data we can calculate the F_1 score. It is important to evaluate the application with previously unseen images to prevent overfitting to the

training data. The data sets used in this chapter are publicly available [SZS14] under the Open Database License.

The GA performs the optimization on integer vector genes. Therefore, we converted all floating point parameters, for instance, the detection thresholds for the virus classification, to fixed point number encoded as integers. We adjusted the fix point precision accordingly to meet the accuracy requirements of the application. This parameter conversion improves the exploration time. Regarding the configuration for NSGA-II algorithm, the crossover was set to a tournament selection with a likelihood of 0.9. Further, the mutation rate was set to 0.1 with a likelihood of 1.0. We let the GA generate 40 generations each with 100 individuals leading to a total of 4,000 solution candidates. For higher accuracy, each execution and thus measurement was repeated 3 times resulting in 36,000 measurements. These settings apply for all three experiments. As already mentioned, we used two Odroid-XU3 boards for a parallel evaluation and the evaluation time for each experiment was around four days. Thanks to the flexibility of our DSE, additional Odroid-XU3 boards can be used to further reduce the run time of the experiments. Furthermore, we observed that the DSE already obtained good solutions after 500 evaluations. Therefore, we derive, that we can further reduce the number of individuals if necessary.

5.5.3 Results

The experiments were assessed regarding the three objectives execution time, energy consumption and detection quality. Obviously, a short run time with minimal energy consumption meeting F_1 score of 1 in detection quality is a desirable solution. In addition, we were interested if the DSE is able to generate a broad front of different solutions. In the following we present the results of the experiments. The figures 5.8 to 5.10 highlight the Pareto frontiers for a more comprehensive representation. In addition, Table 5.2 lists interesting Pareto points on the frontiers.

The baseline experiment Exp0, reported in Table 5.2, shows the performance of an unmodified virus detection software running on the Odroid-XU3. This configuration reaches an F_1 of 1 on the training data and 0.995 on the testing data. The algorithm required 370 Joule and the execution time was 119.8 seconds corresponding to 7.5 fps and thus far from meeting the soft real-time requirement. The frame rate is calculated for 900 images of the data set, as 100 of the 1,000 images are used for the initialization phase. The baseline configuration has been previously optimized on a desktop system regarding the detection quality and was not further tailored towards the Odroid-XU3. On the desktop system execution time and energy consumption was neglected. All solutions already achieved the soft real-time requirements and a (practically) infinite amount of available energy was assumed.

Figure 5.8 shows the results of Exp1_{hw} where only hardware-related parameters without influence in the detection quality were explored. Table 5.2 lists an excerpt of the Pareto frontier. Therefore, the F_1 score does not change and is fixed to 1. As the results show, the execution time ranged from 116.2 seconds (7.7 fps) to 118.9 seconds (7.6

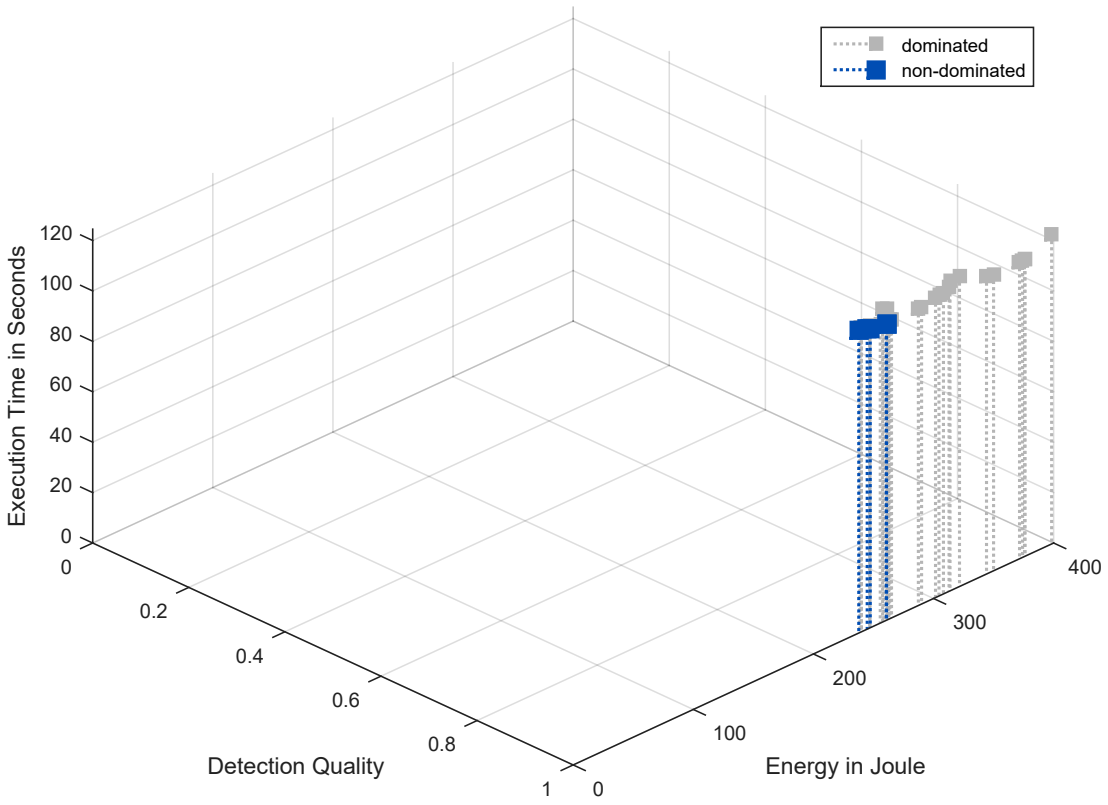


Figure 5.8: Results from Exp1_{hw}. Only hardware-related parameters were optimized, with highlighted Pareto frontier.

fps). The energy consumption offers larger improvements and ranges from 7% to 37% compared to the baseline. Further investigations revealed that the *powersave* governor plays a major role for the slowest but most energy efficient solution. In addition, rather small OpenCL work group and polygon sizes were used. Even though the execution times are still high, at least a notable reduction in energy consumption was achieved.

Figure 5.9 visualizes the results of Exp2_{sw}. Table 5.2 reports an excerpt of the Pareto frontier. In this experiment, the DSE explored only the software parameters of the virus detection software. Already with this experiment we observe that a mobile virus detection using the PAMONO sensor system is possible. The best solution in terms of detection quality ($F_1 = 1$ on the training data) processed all images in 29.7 seconds which corresponds to a frame rate of 30.3 fps. The fastest solution required 10.4 seconds (86.5 fps) with the lowest detection quality on the Pareto frontier. The solution with the best accuracy ($F_1 = 1$) consumed 87.2 Joule. On the other side of the energy consumption spectrum, the optimization algorithm generated a configuration which required 34.4 Joule which leads to the lowest detection quality of 41.3% that is still useful for some detection tasks. As expected, the detection qualities on the testing data set are slightly lower than on the training data set, but still show good results.

Figure 5.10 plots the results of $\text{Exp3}_{\text{hw\&sw}}$ and Table 5.2 includes an excerpt of the Pareto frontier. In this experiment we released the limitations of the DSE and let the algorithm explore both hardware and software parameter simultaneously. Compared

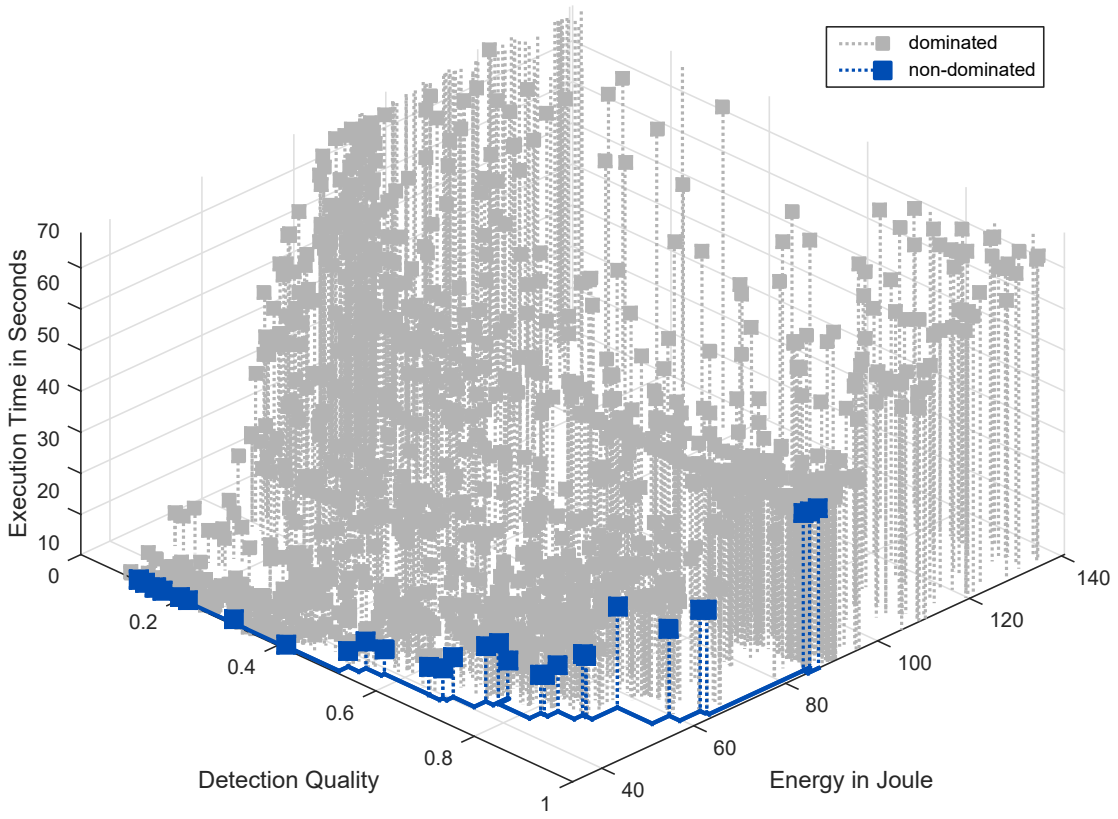


Figure 5.9: Results from Exp2_{sw}. Only software-related parameters were optimized, with highlighted Pareto frontier [NLE15].

to the previous experiments, similar results in detection quality are achieved. However, energy consumption and execution time were further improved. The energy consumption of the best detection quality solution could be reduced by 84% to 57.5 Joule compared to the baseline configuration. Considering the solution with a slightly reduced detection quality of 0.969, in this case, the energy consumption could be reduced by more than 50% compared to the solution from Exp2_{sw} with the detection quality of 0.953 and by 93% compared to the baseline.

5.5.4 Discussion

One indication regarding the quality of the GA and the overall optimization problem is the spectrum of solutions. As the results show, our GA generates a large diversity in the solution space. This can be observed in the structure of the broad Pareto frontiers. Therefore, the user of our framework can select a solution from a set of solutions with different performance characteristics.

Due to the nature of the application, small changes of the parameters could result in solutions where no virus could be detected. For example, if the threshold of the polygon size describing the virus is too high, no viruses are detected. Those results can be observed in Figure 5.10 on the left side.

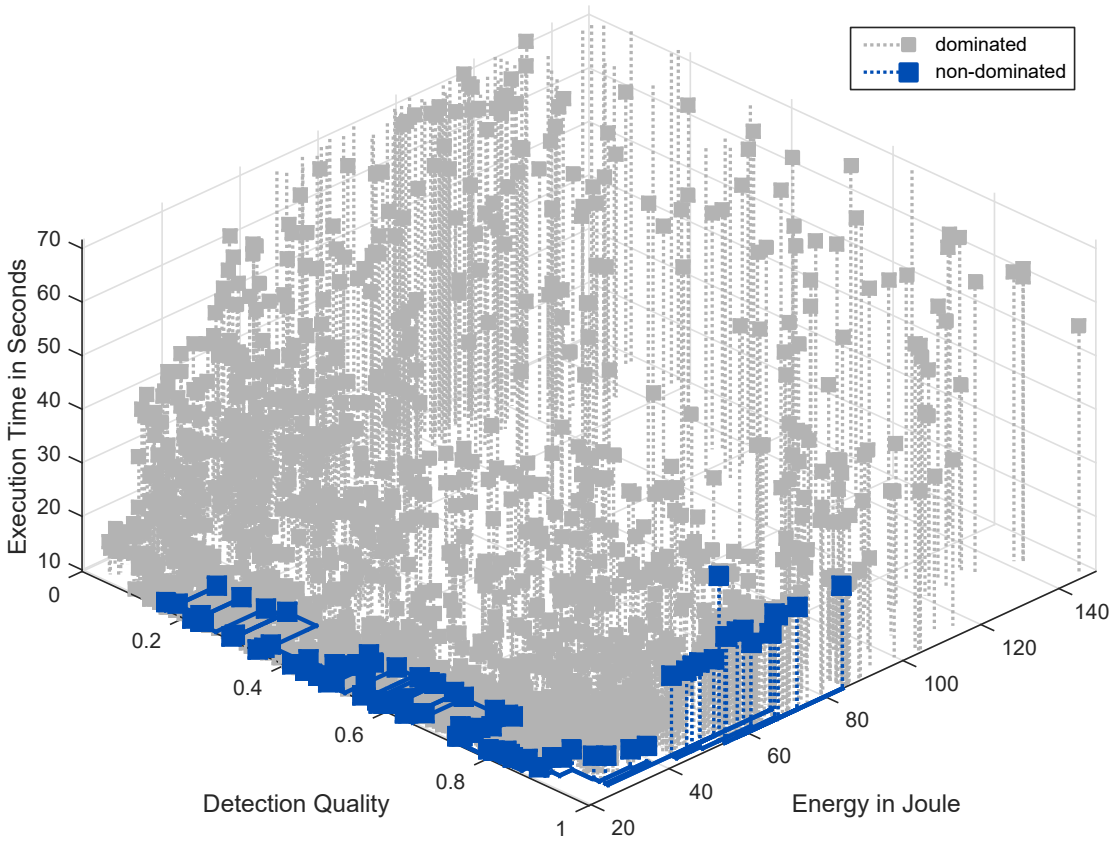


Figure 5.10: Results from $\text{Exp3}_{\text{hw\&sw}}$. Both hardware and software parameters were optimized, with highlighted Pareto frontier [NLE15].

The evaluation lets us conclude that the simultaneous optimization of software and hardware parameters ($\text{Exp3}_{\text{hw\&sw}}$) achieves the best results. The DSE found solutions achieving an F_1 score of 1 on the training data and almost 1 on the testing data running at a frame rate of almost 31 fps. These notable results attest that the detection software meets the soft real-time requirements on the Odroid-XU3 and thus lay the foundation for a mobile virus detection solution. However, for these solutions the energy consumption is comparably high. Accepting a degradation in the detection quality offers huge optimization potential. For example, the energy consumption could be reduced drastically by about 93% if a degradation of $F_1 = 0.878$ is accepted. In addition, this solution boosts the frames per second up to 60.8 fps which is almost doubled. Similar observations can be made for the experiments where the DSE only explored the software parameters in (Exp2_{sw}).

To achieve additional improvements in energy consumption, the degradation of the detection quality must be quite high. Whether such a decrease is useful in clinical practice, highly depends on the use case. If it is only of interest if a sample does contain a certain virus type, then a fairly low detection quality of $F_1 \geq 0.5$ might be sufficient. With this setting only a positive or negative result is the output of the virus test and the result can be accurate even if the actual count of individual viruses is not accurate.

$\text{Exp3}_{\text{hw\&sw}}$ contains several peak shaped solutions on the Pareto frontier. Figure 5.10

Experiment	F_1 Train.	F_1 Test.	Energy	Savings	Exec. Time	Speedup	Frame Rate
Exp0	1.000	0.995	370.0 J	-	119.8 s	-	7.5 fps
Exp1_{hw}	1.000	0.995	233.5 J	37%	118.9 s	1	7.6 fps
	1.000	0.995	239.8 J	35%	117.1 s	1	7.7 fps
	1.000	0.995	246.4 J	33%	116.7 s	1	7.7 fps
	1.000	0.995	257.7 J	30%	116.6 s	1	7.7 fps
	1.000	0.995	344.6 J	7%	116.2 s	1	7.7 fps
Exp2_{sw}	1.000	0.995	87.2 J	76%	29.7 s	4.0	30.3 fps
	0.985	0.931	64.6 J	83%	22.7 s	5.3	39.6 fps
	0.953	0.883	59.7 J	84%	20.6 s	5.8	43.7 fps
	0.870	0.844	50.5 J	86%	17.4 s	6.9	51.7 fps
	0.830	0.734	48.6 J	87%	15.8 s	7.6	57.0 fps
	0.753	0.723	43.9 J	88%	17.4 s	6.9	51.7 fps
	0.750	0.693	46.3 J	87%	14.7 s	8.1	61.2 fps
	0.684	0.612	39.2 J	89%	13.8 s	8.7	65.2 fps
	0.519	0.413	36.4 J	90%	12.0 s	10.0	75.0 fps
	0.413	0.400	34.4 J	91%	10.4 s	11.5	86.5 fps
$\text{Exp3}_{\text{hw\&sw}}$	1.000	0.995	57.5 J	84%	29.3 s	4.1	30.7 fps
	1.000	0.995	84.5 J	77%	28.9 s	4.1	31.1 fps
	0.985	0.974	47.9 J	87%	25.5 s	4.7	35.3 fps
	0.974	0.995	69.3 J	81%	23.9 s	5.0	37.7 fps
	0.969	0.878	27.7 J	93%	14.8 s	8.1	60.8 fps
	0.879	0.766	22.3 J	94%	10.8 s	11.1	83.3 fps
	0.842	0.605	20.7 J	94%	11.4 s	10.5	78.9 fps
	0.742	0.639	23.5 J	94%	10.7 s	11.2	84.1 fps
	0.742	0.647	33.6 J	91%	10.4 s	11.5	86.5 fps
	0.519	0.558	33.0 J	91%	10.0 s	12.0	90.0 fps

Table 5.2: Excerpt of the three Pareto frontiers for the objectives virus detection quality (F_1 score), energy consumption and execution time. In addition, the detection quality (F_1 score testing) for the unseen testing data set is shown. As baseline/comparative measurement an unoptimized run is given in the first row (Exp0), which was measured with an unmodified system and program [NLE15].

shows these peak points with similar detection quality but with differences in execution time or energy consumption. Further investigations revealed that different governor settings led to this behavior. More energy efficient solutions used the *userspace* governors with a fixed frequency of 1.1 GHz for the Cortex-A15. Using *conservative* or *performance* governors on the other side resulted in slightly faster solutions that consume more energy.

In general, the results indicate that the energy consumption is not always tightly coupled to the execution time. For instance, Exp1_{hw} shows that changing only the hardware parameters can influence the energy consumption in the opposite direction to the execution time. $\text{Exp3}_{\text{hw\&sw}}$ shows another interesting solution, here, a slightly faster solution (86.5 fps, 33.6 Joule) consumes 42% more energy than the slightly slower (84.1 fps, 23.5 Joule) one. Further investigations show that the use of different memory

access patterns led to these results. However, in many cases, execution time and energy consumption are related and thus applying improvements to one of the two often benefits the other as well.

To summarize, with the experiments we showed that the DSE is able to explore the parameter space of complex applications efficiently and that it generates good solutions targeting embedded systems. For the analyzed virus detection software, the optimization algorithm was able to improve the execution time drastically and still achieved a high detection quality. In this case, the results showed a speedup of 4.1 with an energy saving of $77\% - 84\%$ with an F_1 score of 1. We already discussed the capability to accept a slight degradation in the detection quality under certain circumstances. In such a case, we observed an impressive speedup of 8.1 and energy savings of 93% by only sacrificing only small amount of detection quality. This solution still achieved a fairly good detection quality of 0.969. Leveraging the detection quality even more reveals solutions with a speedup of 12 and energy savings of up to 94%. To conclude, the results verify that a real-time mobile virus detection solution using the PAMONO sensor is already feasible with current very reasonable embedded systems.

5.6 Related Work

The PAMONO sensor setup and the virus detection software are unique, therefore this section presents related work regarding design space exploration. Our work extends traditional (hardware-oriented) DSE by considering detection quality as an additional objective. The evaluation shows that leveraging the detection quality provides additional improvements. In general, this technique is known as approximate computing and this thesis dedicates a separate chapter (cf. Chapter 6) to this research field. Hence, we will present additional related work there.

Embedded systems face new challenges exposed by complex application which should be executed efficiently on them, desirable in a mobile battery driven system. Even in stationary applications, the available energy or related thermal budgets might be limited. For instance if a system produces too much thermal heat ($\approx$ power consumption), the connected sensor may produce faulty data or fail entirely. Therefore, it is obvious that energy consumption and computation power are potentially contradicting although they are key objectives. In this chapter we analyzed an application from the computer vision and machine learning domain where the GPU plays an important role and contributes significantly to the overall energy consumption. Mittal and Vetter [MV14] presented a survey considering GPU energy efficiency. They found that for battery powered devices the need for performance requires aggressive and sophisticated energy optimizations. Cebrian et al. [CGG12] analyzed the energy efficiency of desktop GPUs and conclude that the energy consumption heavily depends on the actual application. Therefore, to achieve necessary efficiency for mobile usage, energy-aware computing is mandatory. We support this proposition with our insights gained by this work. In our case, optimizing

the hardware-related parameters alone was not sufficient to improve the application in such dimensions as optimizing them together with the software parameters. Ahmad and Ranka [AR12] did a survey regarding energy-aware computing in general.

Several Design Space Exploration (DSE) approaches assisting in finding good solutions in huge parameters spaces have been presented in the last decades. Pimentel [Pim17] provides an introduction into DSE especially targeting the embedded systems domain. The evaluation in this chapter required almost four days which highlights that, in this case, most DSE's run time is caused by the evaluation time of the solution. Pham et al. [PSK13] presented a combined energy and throughput aware approach using online and offline exploration targeting heterogeneous MPSoC. The key idea is that an offline generated solution is refined during the execution time. Their DSE approach only considers mapping and no hardware or software parameters and neglects opportunities offered by leveraging QoS. The hardware/software co-exploration approach of Agosta et al. [APS04] explores architectural parameters and source-level code transformations concurrently to find trade-offs between energy consumption and delay (execution time) targeting low-power embedded applications. Here, hardware parameters are for instance memory hierarchy levels. Source-level transformations include function inlining and loop unrolling. This approach does not consider QoS and only uses two software transformations and thus may waste opportunities exposed by the application parameters. The MADNESS framework [CGF11] also provides a DSE to explore multiple objectives simultaneously. This approach is based on the DAEDALUS framework [NTS08] and is able to compose an entire MPSoC using a library of predefined hardware blocks and software implementations. To prune the parameter space, application scenarios are used. In such a scenario, multiple applications are bundled and handled as one entity. In addition, different scenarios can capture specific implementations of the same application to take, e.g., different QoS into account.

Palesi and Givargis [PG02] developed a multi-objective aware approach to map software onto configurable SoC. Their GA-based DSE considers timing requirements and power budgets. Using model-based DSE promise to reduce the exploration time. The Octopus toolset [BVG10] maps tasks to processors, GPU, memories and Field Programmable Gate Array (FPGA) components with the objective to maximize the throughput of the application. Their approach only uses abstract models and requires a profound interaction of the user.

Exploring heterogeneous embedded systems with a hierarchical approach was presented by Mohanty et al. [MPN02]. Their rapid search algorithm starts at a high abstraction level and uses functional simulators to verify the functional correctness. High-level performance and power estimators prune the design space. Promising solutions are then evaluated with a precise cycle accurate simulator.

Keinert and Teich [KT11] designed a DSE focusing on image processing applications. Their approach targets embedded system and their composition. Starting with a data flow description of the application, the approach leads to a synthesized embedded system. However, their approach neglects GPUs although they are well suited for image processing.

An energy-aware measurement-based DSE targeting GPGPU applications was presented by Park et al. [PKS13]. Their approach explores task mapping onto the target GPU. The DSE as only considered two hardware related parameters.

5.7 Conclusion and Future Work

Traditional methods to detect nano-sized objects, especially viruses, are time-consuming and require special equipment and trained personal. In cases of disease control, the time between contamination and the ability to detect the virus is crucial and should be short. The PAMONO sensor method uses an indirect optical method and provides the fundamentals for a real-time virus detection. A virus detection software detects and counts viruses in images captured by the sensor using sophisticated image processing algorithms. With the Odroid-XU3, we investigated the possibility to run such sophisticated application on an embedded system.

We extended an existing DSE towards embedded systems and used the Odroid-XU3 as an exemplary target platform. The DSE framework is now able to explore hardware and software parameters simultaneously and evaluate solution's performance on a wide variety of systems. Despite previous version of the DSE algorithm, execution time and energy consumption as well as detection quality were considered by the DSE which should provide additional optimization opportunities especially for resource restricted embedded systems.

With three experiments, we evaluated the performance influences of hardware and software parameters and applicability of the enhanced DSE for these scenarios. We were able to find solutions meeting the real-time requirements without leveraging the detection quality. This impressive result was initially not expected and proves the capabilities of modern embedded systems combined with our advanced optimization framework. Compared to the baseline, the combined hardware/software exploration found solutions with a speedup of 4.1 leading to a frame rate of 30.7 fps and an energy consumption of 57.5 Joule which translates to a saving of 84%. Accepting a small degradation in the detection quality, which can still be usable in several medical contexts, leads to solutions with speedups between 8.1 and 11.1. Thus, the frame rate exceeds the real-time requirements almost three times. Therefore, we conclude that accepting inaccurate results can help to improve the performance.

In the current state, the PAMONO sensor only allows to detect a stem of viruses which attach to the same type of antibody. To increase the detection range, a (larger) gold layer can be partitioned and coated with different antibodies. The detection process gets complicated with each additional virus type but we showed that some performance headroom is available. In this case, the image resolution must be adapted to the new gold layer size leading to a higher computational demand. Instead of leveraging the detection quality, the DSE framework could be extended to support independent optimization for separate regions in the recorded image. In such case, the optimization algorithm

might increase the detection quality in some areas whereas reducing the quality in others. Instead of detecting multiple virus types simultaneously, the available headroom can be used to increase the resolution of the images and still meet the real time and energy requirements. Finally, a cheaper hardware system might be sufficient to achieve the virus detection task.

The insights obtained in this chapter especially regarding the detection quality trade-offs lead us to an interesting research question: can techniques of approximate computing further improve PICO?

Chapter 6

New opportunities due to Approximate Computing

Contents

6.1 Introduction

Growing demand for processing power, especially in the high-performance embedded systems application domain such as autonomous cars and media signal processing, increases the pressure on hardware and software developers to create well performing systems which are also power and energy efficient. Traditionally, hardware manufacturers tackled the performance challenge by increasing the frequencies of processors and memories, whereas the reduction of power and energy consumption was mostly achieved by shrinking semiconductor structure sizes. However, frequency increase is limited by thermal and

energy constraints and parallelism suffers from synchronization overhead (cf. Chapter 4). These so called performance and power walls expose a serious challenge for the growing performance demand. Several techniques have been developed to push these walls away such as the resource-aware parallelization methods developed in the context of this thesis. To deal with *dark silicon* [EBA11], other approaches try to integrate deep sleep states or power gating of entire parts of modern processors into their software algorithms in order to keep energy and thermal budgets within certain limits.

One increasingly popular method to mitigate the rising impact of these performance and power walls is to use *approximate computations*. This paradigm softens the QoS requirements of an application by accepting solutions which are not bit-perfect. Traditionally, recognition, data mining and search (RMS) algorithms use this paradigm to stop their calculations if certain termination thresholds are met. However, the main idea behind approximate computing is to use less accurate computations in hard- or software to gain improvements in energy consumption or performance. Unfortunately, there is no free lunch, so the resulting output usually is less accurate. The *lossy encoding* technique follows a similar idea and is exploited especially in the media domain to reduce file sizes and throughput requirements. We define approximate computing as follows:

Definition 6.1 (Approximate Computing):
Approximate computing allows a certain deviation from the perfect result in order to improve in other performance objectives.

In this thesis, we focus on software approximation techniques. A common approach is to provide several implementations of the same algorithm with different accuracy levels. Other examples are the reduction of the precision of data types, computation loops may skip iteration steps or parallel applications may relinquish of certain synchronization points. The user then selects the suiting implementation. However, in various cases the selection criteria and the impact onto the entire application and the overall performance is unclear.

Resource constraints are always a key concern of embedded systems by nature, thus approximate computing could be a very promising way to gain additional performance improvements. In addition, another important advantage is the fact that approximation techniques enable applications to be executed on embedded systems which otherwise lack of sufficient computation power. In Chapter 5, we successfully demonstrated that leveraging the detection quality of the PAMONO virus detection pipeline offers huge performance benefits whereas it is still usable for mobile medical applications. With that work we successfully mapped the virus detection software onto a mobile embedded system.

The next step is to move towards an areal monitoring system with multiple sensors placed at public locations like airports. To achieve a certain cost-effectiveness, it might be necessary to split the data sampling and processing task. In such a case, fairly inexpensive low-power embedded systems take the sample images and send them to a central powerful computer to evaluate the data streams. To achieve this, parts of the

virus detection pipeline must be adapted to these low-power systems and approximation might be key to achieve this.

Finally, in the traditional CPS domain the interaction with noisy sensors and actuators has always been an important aspect. Internally these algorithms map those inherently noisy inputs onto high-precision data types to perform a costly but precise calculation. Consequently, working with uncertainty inside applications of CPS might improve overall performance.

An efficient parallelization of a sequential application with respect to resource restrictions is the goal of the PA4RES project. Therefore, approximate computing might offer additional benefits. The quantification of the impact of approximation onto applications is a complex task and done improperly might lead to false conclusions. Hence, this chapter presents a (semi) automatic assessment framework to analyze the influence of approximation techniques onto the application performance and output quality.

The Quality Comparison for Approximate Programs on Embedded Systems (QCAPES) enables an automatic evaluation of source code level approximation techniques with respect to various quality metrics. This is particular useful for a safe introduction of approximation on a software level into existing applications and the development of new software approximation techniques. In addition, using multiple metrics might provide a deeper understanding of the applied approximation impacts. With two case studies, we demonstrate that the selection of the correct quality metrics and objectives is crucial especially in the embedded domain. The studies revealed that a popular approximation technique applied carelessly to the embedded domain renders all benefits useless. With this knowledge, we extended PICO and the entire PA4RES methodology towards approximation computing techniques.

This chapter includes work published in [NMK17], [Küh16] and [NEM15a]. Approximate computing has been used in several applications and Section 6.2 presents important related works. Quality metrics are used to quantify the impact of approximation on the output. Section 6.3 presents commonly used metrics and highlights some pitfalls using them improperly. The Quality Comparison for Approximate Programs on Embedded Systems (QCAPES) framework is presented in Section 6.4. With two case studies, we demonstrate in Section 6.5 the capabilities of QCAPES to support the developer during the introduction of approximation techniques into their applications and reveal unknown side effects of a careless usage of approximation with respect to embedded systems. With this knowledge Section 6.6 presents extensions introduced to PICO in order to support approximation. Finally, Section 6.7 concludes this chapter and provides directions for future work.

6.2 Related Work

Approximate computing promises to be a worthwhile approach to overcome or at least shift the influence of the performance and power walls. In recent years, related techniques

have seen significant adoption [Kug15; XMK16; HO13; VCR15b; VCR15a; Mit16]. Starting with examples of hardware-based approaches, we focus on the relevance of applying quality metrics during the development of software-based approaches.

On the hardware side, approximate computing can be applied to almost all components involved in the calculation. For instance, approximate adders trade accuracy with execution speed by accepting inaccuracies in the least significant digits of a calculation [KMM10]. Several techniques, e.g. reduction of the supply voltage and sub-threshold operation were developed to reduce the energy consumption by sacrificing numerical precision. Reducing the supply voltage for the inaccurate part will reduce the energy consumption by accepting calculation errors in this part. Another way is the logic complexity reduction at the transistor level [GMP11]. Reducing the refresh rate of memories reduces the energy consumption, but stored data might be erroneous [LJV12; LPM11]. The MACACO project [VAR11] focuses on an efficient modeling and analysis of approximate hardware.

Software approaches are usually more flexible than hardware ones since they can be adapted to specific scenarios or dynamically adjusted at run-time. Application knowledge can be introduced, exploring the specific requirements and capabilities of a given piece of software. In principle, software approximation techniques can be executed on commodity hardware and added to existing systems. In addition, a software approach avoids the problem of obtaining unexpected results from inaccurate hardware computations and the problem of verification of inaccurate hardware components.

Software approximation in general can be applied to programs where a result is refined using loops. As an example, iterative calculation can be stopped after the result meets certain requirements. Omitting iterations is the basic idea behind *loop perforation* [MSH10; SMH11]. Changing the precision of numeric values, e.g. from *float* to *int* can also lead to performance improvements. Using approximate data types and operations are the key features of EnerJ [SDF11]. Through source annotations, EnerJ declares data or operations to be subject to approximations.

The GREEN system [BC10] uses controlled approximation to reduce energy consumption by meeting a certain QoS. The user provides multiple implementation of an algorithm with different precision and e.g. energy consumption. Then, the GREEN system builds a QoS model to determine the impact of the provided implementations on the QoS. Afterwards, a solution is selected which meets the user-specified QoS requirements. Thus, a trade-off between energy consumption/performance and QoS is made. Nevertheless, the user has to provide several inputs like implementation or QoS which is complex and not always possible. The SAGE approach [SLJ14] combines automatic code generation, to generate various levels of approximation, with a runtime system to achieve speedups under user-defined output quality requirements. This approach concentrates on GPUs only as target platform and trade-offs between speedup and output quality.

The Dynamic Knobs approach [HSC11] includes a feedback loop and adjusts the internal approximation algorithms regarding the output precision during the execution

time. Parallel applications can suffer from large synchronization overhead. To reduce the waiting time, [RSN12] proposed a relaxation of the synchronization barriers by keeping the output quality in an acceptable range.

The presented approaches either rely on user-defined or heavily application-dependent quality metrics. In most cases, only one metric is considered. Akturk et al. [AKK15] presented hints on metric selection for different domains. However, in general, choosing the appropriate metric during the development process is a sophisticated task. The following sections present details on metric selection and demonstrates pitfalls of a careless utilization of approximate computing.

6.3 Quality Metrics - How to quantify uncertainty?

Quality metrics quantify the impact of approximations on the output's accuracy or applications' QoS. In the last decades, various metrics to quantify the output quality have been proposed. Each domain has a set of preferred metrics. Therefore, a good metric selection can be complex and error-prone, especially during the development of application-agnostic software approximation approaches.

In general, metrics can be partitioned in two classes. The first class evaluates the output regarding mathematical properties. These metrics solely use the output signal and neglect the final application usage to quantify the result quality. Therefore, this class is called Signal Fidelity Metrics (SFM). These metrics are generally fast to calculate and thus useful for an online QoS management and they saw a broad adoption in various domains.

The second metric class takes the application scenario into account and might be better suited to quantify the quality for a given application. The F_1 measure (cf. Subsection 5.4.2) calculates the ratio between training and test data and is very application specific. In general this class of perception-based metrics takes the receiver into account and thus provides a perception model. In this chapter we focus on applications from the image processing domain. Accordingly, the class is called Perceptual Visual Quality Metrics (PQVM).

6.3.1 Common Signal Fidelity Metrics

Signal Fidelity Metrics (SFM) solely rely on the signal itself to calculate the quality. SFM are typically fast to calculate and applicable in many domains. In the context of approximate computing, we are interested in the difference between the approximated and the original (perfect) result. Let $x_0, ..., x_n$ and $y_0, ..., y_n$ be series of (discrete) signal samples whereas y_i is the result of the approximated version and x_i represents the perfect or best version. To compute the quality of the approximated version compared to the original value, we introduce in the following some commonly used metrics.

The most simple metric is the Mean-Squared Error (MSE):

Definition 6.2 (Mean-Squared Error (MSE)):

$$MSE(x,y) = \frac{1}{n}\sum_{i=0}^{n}(x_i - y_i)^2 \tag{6.1}$$

Ideally, the MSE should be zero or close to zero. The Root-Mean-Squared Error (RMSE) extends the MSE:

Definition 6.3 (Root-Mean-Squared Error (RMSE)):

$$RMSE(x,y) = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(x_i - y_i)^2} \tag{6.2}$$

Similar to the previous metrics, Mean-Absolute Error (MAE) calculates the true mean error:

Definition 6.4 (Mean-Absolute Error (MAE)):

$$MAE(x,y) = \frac{1}{n}\sum_{i=0}^{n}|x_i - y_i| \tag{6.3}$$

The presented metrics are able to calculate a relative deviation. If the errors of each sample are nearly equal the RMSE and the MAE will produce a similar result. But in contrast to the MAE, the RMSE is more sensitive to big outliers [CD14]. Since the RMSE is limited downwards by the MAE and upwards by $\sqrt{number\ of\ samples} \cdot MAE$ [WM05] both metrics can be used to obtain a better understanding of the errors.

The Peak-Signal-to-Noise Ratio (PSNR) makes results better comparable, due to the inclusion of the maximum signal range x_{max}. Furthermore, this enables the comparison of signals with different value ranges like bit-depths:

Definition 6.5 (Peak-Signal-to-Noise Ratio (PSNR)):

$$PSNR(x,y) = 10\log_{10}\left(\frac{x_{max}^2}{MSE(x,y)}\right) \tag{6.4}$$

$$= 20\log_{10}\left(\frac{x_{max}}{RMSE(x,y)}\right) \tag{6.5}$$

The SFM are usually fast to compute and describe the mathematical deviation from the given (perfect) result. However, these metrics do not provide any information what effect such deviation has on the final application. Furthermore, in many cases, the nature of the errors can not be detected [WB09]. Therefore, the following subsection presents two perception-based metrics.

6.3.2 Perception Visual Quality Metrics

Perception-based metrics are constructed in such a way that the impact on the receiver side is modeled. Humans are usually more sensible to errors in specific value ranges. This effect is exploited in *lossy encodings* which remove parts of information humans usually do not perceive or at least are very insensitive to. For instance, the prominent MP3 audio encoder removes very high frequencies from the audio sample to reduce the file size, as most people can not hear them. In the context of image processing application, the usual recipient is the human visual system (HVS). Therfore, the Perceptual Visual Quality Metrics (PQVM) incorporates HVS characteristics in the evaluation of images.

The HVS is very sensitive to structural changes and metrics should consider for this. In contrast to SFM, the Universal Image Quality Index (UIQI) [WB02] is designed to work on grayscale images or the Y-component of a color image. The UIQI consists of three different factors, which examine different characteristics of an image. The luminance component l compares the luminance of two images:

$$\mu_x = \frac{1}{n}\sum_{0}^{n} x_i \tag{6.6}$$

$$\mu_y = \frac{1}{n}\sum_{0}^{n} y_i \tag{6.7}$$

$$l(x,y) = \frac{2\mu_x\mu_y}{\mu_x^2 + \mu_y^2} \tag{6.8}$$

The second component measures the difference in contrast c of two images:

$$\sigma_x = \sqrt{\frac{1}{n-i}\sum_{i=0}^{n}(x_i - \mu_x)^2} \tag{6.9}$$

$$\sigma_y = \sqrt{\frac{1}{n-i}\sum_{i=0}^{n}(y_i - \mu_y)^2} \tag{6.10}$$

$$c(x,y) = \frac{2\sigma_x\sigma_y}{\sigma_x^2 + \sigma_y^2} \tag{6.11}$$

and the third component determines the loss of correlation s between the two examined pictures:

$$\sigma_{x,y} = \frac{1}{n-1}\sum_{i=0}^{n}(x_i - \mu_x)(y_i - \mu_y) \tag{6.12}$$

$$s(x,y) = \frac{\sigma_{x,y}}{\sigma_x\sigma_y} \tag{6.13}$$

Finally, the UIQI is the product of l, c and s.

Definition 6.6 (Universal Image Quality Index (UIQI)):

$$Q(x,y) = l(x,y) \times c(x,y) \times s(x,y) \tag{6.14}$$

The value range of UIQI lies within an interval of $[-1,1]$; higher values represent a better quality. Since details lose their significance if the UIQI is applied globally, it is just applied on small local windows, typically 8×8 pixels. Normally, a sliding window approach where the window moves pixel by pixel across the entire image is used to calculate the overall quality.

Due to the fact that the UIQI is based on statistical procedures like arithmetic mean, standard deviation and correlation coefficient, the UIQI tends to be unstable if all values in a window are nearly equal. The Structural Similarity Index Metric (SSIM) [WBS04] differs in the three single components from the UIQI and adds constants C_i to each factor to obtain stable results. In addition, each component can be weighted with the exponents α, β and γ. The SSIM for a pixel block is defined as:

Definition 6.7 (Structural Similarity Index Metric (SSIM)):

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \qquad c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, \qquad s(x,y) = \frac{\sigma_{x,y} + C_3}{\sigma_x\sigma_y + C_3}$$

$$SSIM(x,y) = l(x,y)^\alpha c(x,y)^\beta s(x,y)^\gamma \tag{6.15}$$

The Mean Structural Similarity Index Metric (MSSIM) across the entire image is defined as:

Definition 6.8 (Mean Structural Similarity Index Metric (MSSIM)):

$$MSSIM(x,y) = \frac{1}{n}\sum_{i=0}^{n} SSIM(x_i, y_i) \tag{6.16}$$

To improve the expressiveness of the metric if blocking artifacts occur (e.g. JPEG compression) a 11×11 sliding window with a Gaussian filter is proposed. One reason why we use the UIQI as well is pointed out e.g. by Egiazarian et al. [EAP06]. In their paper, they demonstrated that the SSIM value of heavily distorted images can be misleading while the UIQI reports useful information. It has been shown in [HZ10] that the SSIM is more sensitive to JPEG compression errors, while the PSNR is more sensitive to Gaussian noise.

6.3.3 Impact of Metric Selection

So far, we presented six different metrics to quantify the quality of the applied approximation technique. In the following, we stress the importance of the metric selection for a safe introduction of approximate computing into existing applications. The metric selection is even more important during the development of new approximation techniques to get early insights into the consequences on the quality.

(a) Original (b) MSE = 460, SSIM = 0.95 (c) MSE = 459, SSIM = 0.50

Figure 6.1: Quality Comparison according to MSE and SSIM [NMK17].

Figure 6.1 shows three versions of the same images whereas 6.1(a) is the original image. We brightened the image in 6.1(b) and added a Gaussian blur to 6.1(c). With a SFM like MSE it is almost impossible to distinguish which images of the two modified is *better* since the values are almost identical. Fortunately, PQVM like SSIM provide more insights and let us conclude that 6.1(b) is *better* then 6.1(c) for human receivers. This observation can also be applied to pattern recognition algorithms, for instance, an edge detection algorithm should perform better on the image shown in 6.1(b) since it preserves the sharp edges. To conclude, this example highlights the importance of the metric selection, and that for a safe introduction of approximate computing, multiple metrics should be considered.

6.4 QCAPES-Framework

The QCAPES assessment framework supports developers during the introduction of approximate computing into their application. In addition, QCAPES provides additional insights into the structure and impact on the quality degradation and performance improvements caused by approximation. The key idea of QCAPES is to execute (multiple) approximated versions and the original application and calculate the impact onto output and performance. The framework already provides several metrics and input decoders like a PNG, JPEG or video decoder focusing on the image processing domain. Furthermore, QCAPES provides decoders for numeric values. In addition, the framework provides support to consider energy consumption, run time and user-defined objectives like file size. The framework is implemented in C++ and comprises roughly 15.000 SLOC including code the for the input decoders. A clear interface design hides complexity and thus

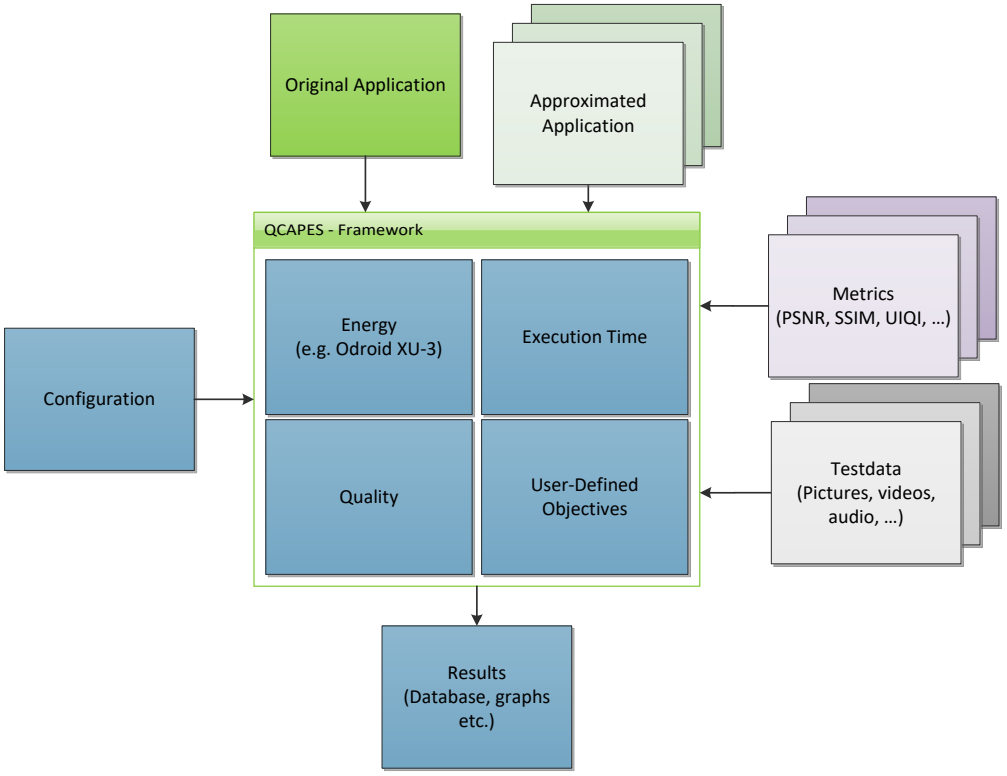


Figure 6.2: QCAPES overview.

enables easy integration of new metrics into the system. The QCAPES framework is available under an open source license [SFB17b].

Figure 6.2 illustrates the internals of QCAPES. The user provides an executable without modifications which output is considered as the reference. Further, the user passes one or several inaccurate versions of the application to the framework. With a configuration file, the user selects metrics, objectives and data sets which are considered for the evaluation. Evaluation results are stored in a database and the framework generates graphs to visualize the execution time, energy consumption and quality results.

Metrics Class	Metric
Signal Fidelity Metrics (SFM)	Mean Squared Error (MSE) Mean Absolute Error (MAE) Root Mean Squared Error (RMSE) Peak-Signal-To-Noise-Ratio (PSNR)
Perceptual Visual Quality Metrics (PVQM)	Universal Image Quality Index (UIQI) Structural Similarity Index (SSIM)

Table 6.1: Overview of included metrics. Several PQVM metrics implementations are provided, e.g. different block size.

QCAPES includes several metrics and Table 6.1 lists them. For SSIM and UIQI, the framework provides several configurations regarding the block size and the sliding window steps. It provides implementations for a block size of 4×4, 8×8 or 16×16 pixels used for the local quality evaluation. During the evaluation process the window is moved

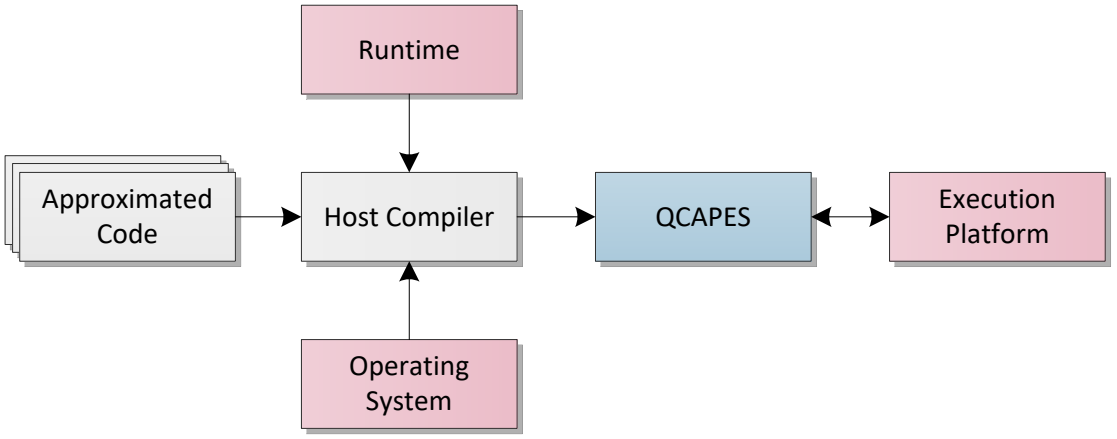


Figure 6.3: PA4RES manual QCAPES flow.

across the input image and the results are combined to indicate the global quality. The window moves across the image either overlapping pixel by pixel or non-overlapping using blockwise steps. Finally, for the SSIM, a Gaussian filter can be used to reduce the influence of pixels at the edges. Thus, in total QCAPES currently includes four SFM and 13 PQVM. In addition, the framework considers color components separately, hence it is also possible to analyze color components as proposed in [WLB04].

6.4.1 Integration into PA4RES

The QCAPES framework can either be used as a standalone application or as an integrated part of the PA4RES tool flow. The standalone version is tailored towards the Odroid-XU3 (cf. Figure 2.3) and uses the EnergyMeter or Energy Relay Reader (cf. Section A.2). QCAPES is either running on the device under test or on a separate host. In the second case, QCAPES issues a remote execution and collects the output data from the test device and performs the output analysis on the host.

QCAPES also provides interfaces to the PA4RES tool flow and thus integrates seamlessly following the MACCv2 approach. Figure 6.3 illustrates the manual approximation flow provided in the PA4RES framework. In this case, the user provides the approximated source code to the framework. The integration into PA4RES allows QCAPES to access the simulation-based platforms as well as additional capabilities offered by the seamless interaction between other MACCv2 tools. Consequently, other tools in the PA4RES framework can access the results of QCAPES.

6.5 Qualitative Case Studies

This section presents two case studies to emphasize the importance of metric and objective selection especially for resource-restricted systems. Furthermore, the studies demonstrate QCAPES's capabilities to support developers during the evaluation of approximate computing techniques and the advantage of a multi-metric assessment. The studies cover two representative applications typically found in the mobile systems domain and various

approximation techniques. The first study applies *loop perforation* to a video encoding algorithm. The second study considers algorithmic choice to leverage output quality in order to gain performance improvements during image compression. All experiments were executed on the Odroid-XU3. To observe the effect of the heterogeneous architecture, we repeated each experiment on the Cortex-A15 and Cortex-A7.

6.5.1 Approximated Video Encoding

In this case study we investigate the *loop perforation* technique applied on the `x264-encoder`[1] [Vid16] as presented by Misailovic et al. [MSH10; SMH11]. This study replicates the original one but focuses on a multi-metric evaluation, the original study used PSNR as the main metric. In addition, our study enhances the original results with energy consumption values and applicability for embedded systems.

We used *perf*, a Linux performance measurement tool, to validate the loop selection of the original paper since an older version of the `x264-encoder` was used in that study. We identified two functions (`x264_pixel_sad_x3_8x8` and `x264_pixel_sad_x3_16x16`) with run time critical loops which might benefit from loop perforation. According to the original study, we choose to perforate the loops with a rate of 50% and 75%. This results in an execution of every second (50%) or forth iteration (75%) respectively. Four videos were used to evaluate the approximated video encoding. Table 6.2 lists the properties of the test videos taken from the PARSEC benchmark suite [BKS08] and the Xiph.Org Foundation [Xip18].

Video	Frames	Dimension	Size
eledream 32	32	640×360	11 MB
eledream 128	128	640×360	43 MB
coastguard	300	352×288	44 MB
crew	300	352×288	44 MB

Table 6.2: Input videos for approximated video encoding case study.

All experiments were executed on the Odroid-XU3. QCAPES managed the entire evaluation process. Figure 6.4 reports the execution time and energy consumption. The values show the percentage run time and energy consumption compared to the original encoding run, thus, lower values are better. As expected, the execution time reduces with an increased perforation rate. These results indicate that approximation indeed reduces the run time and consequently improves the energy consumption.

QCAPES supports the developer during the assessment of the output quality with a multi-metric evaluation. In the following, we focus on the Y-component of the YCbCr color coded video signal since humans are most sensible to this channel. However, QCAPES can also be applied to the other components. Table 6.3 lists the averaged SFM values for all video encoding experiments. The averaged SFM values are similar for both perforation rates, therefore, Figure 6.5 depicts the results of the frame-by-frame analysis

[1] x264-snapshot-20160203-2245

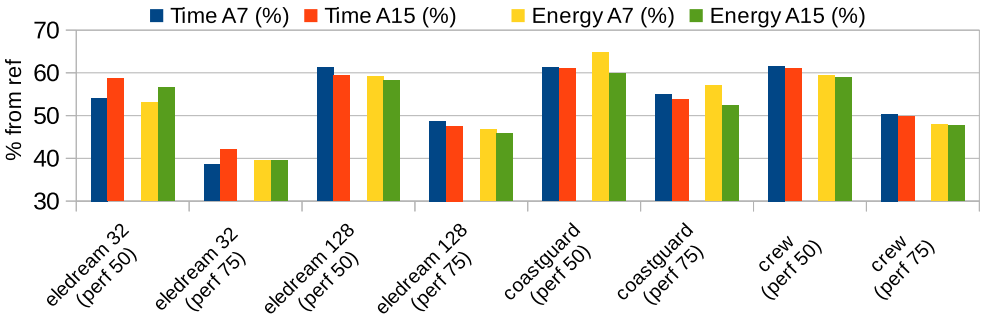


Figure 6.4: Run time and energy consumption for the approximated video encoder.

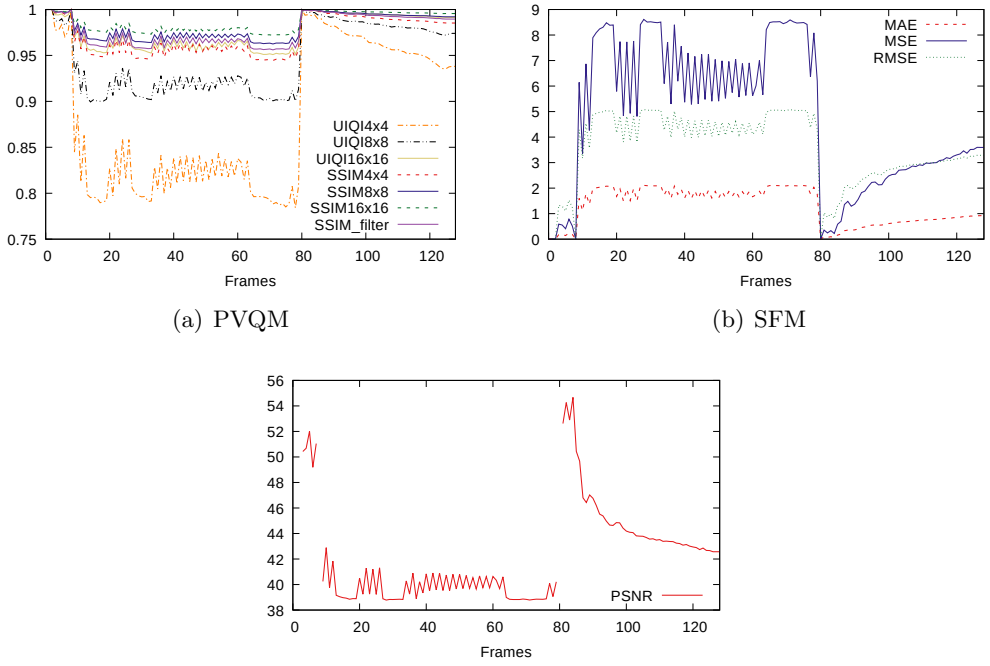
Benchmark	Perforation	MSE	RMSE	MAE	PSNR
eledream 32	50%	5.33	2.05	1.39	41.58
eledream 32	75%	5.15	2.03	1.37	41.53
eledream 128	50%	4.84	2.05	1.26	42.22
eledream 128	75%	4.64	2.00	1.23	42.47
coastguard	50%	8.17	2.83	2.15	39.03
coastguard	75%	8.80	2.94	2.25	38.69
crew	50%	5.93	2.41	1.76	40.44
crew	75%	6.06	2.44	1.79	40.33

Table 6.3: Average signal fidelity for the approximated video encoder.

for the eledream 128 testcase with a perforation rate of 50%. The user can use these data provided by QCAPES for a precise analysis of the impact of the approximation. For PQVM and PSNR, higher values are better, whereas for all other SFMs, smaller values are better. Especially striking are the values of UIQI where the window size has a large influence on the results. The outcome of UIQI on smaller windows is slightly worse than on bigger windows. This is due to a weakness of UIQI that if a window contains identical values the standard deviation of the observed window is zero. Consequently, the correlation coefficient can not be calculated and the window is omitted for determining the overall image quality. If the observed window is larger (e.g. 16×16 pixels) this effect is more unlikely to arise.

The results indicate that *loop perforation* indeed increases the performance by keeping the deviation from the original in a acceptable range. In addition, the results let us derive that the PQVM with smaller block sizes also provide meaningful information. In general, a larger block size reduces the effect of local distortions and therefore results in more precise metric values. If larger blocks produce more precise results, which benefit offer smaller block sizes?

Figure 6.6 visualizes the execution time of the metrics applied to the 128 frame eledream test video. The results verify that SFM and PQVM with smaller block sizes are faster to evaluate. Furthermore, the sliding window implementations that move blockwise (entries succeeded with a b) are also fast to evaluate. The execution time results let us conclude that even on the Odroid-XU3 a run time monitoring of the output



(a) PVQM

(b) SFM

(c) PSNR

Figure 6.5: Frame-by-frame analysis of the Y channel for eledream 128. Higher values in 6.5(a) and 6.5(c) are better, and in 6.5(b) lower values are better [NMK17].

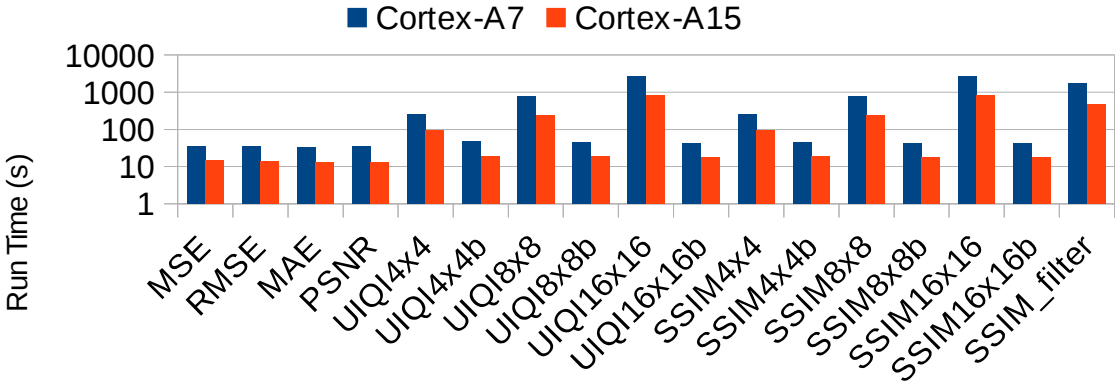


Figure 6.6: Run time for metrics evaluation on eledream 128.

quality is feasible. This enables a future run time adaption of the approximation.

For resource-restricted embedded systems, the resulting file size is of crucial importance. Therefore, either for storing the data or, in case the data must be transmitted over wireless radio channel, an increased file size might render all achieved run time and energy improvements useless. Table 6.4 lists the resulting file sizes of the video compression algorithm. The file size increased across all experiments between 3.1% and 20.1% compared to the resulting size of the reference encoder.

To get an understanding why the file size increased, Table 6.5 shows the resulting frame types of the different perforated versions. A detailed analysis of the frame types for the eledream 128 video shows that increasing the perforation rate increases the amount

Scenario	eledream 32	eledream 128	coastguard	crew
Reference	0.92 MB	2.76 MB	3.26 MB	2.72 MB
50% perforated	0.96 MB (+4.2%)	2.89 MB (+ 4.6%)	3.41 MB (+ 4.6%)	2.81 MB (+ 3.1%)
75% perforated	1.11 MB (+20.1%)	3.17 MB (+ 15%)	3.50 MB (+ 7.4%)	3.08 MB (+ 13.1)

Table 6.4: File size of encoded videos with a perforation rate of 50% and 75%.

of P- and decreases B-frames. P-frames contain more information compared to B-frames leading to the increased file size. Further investigation revealed that by applying *loop perforation* to this compression algorithm the actual compression is partially skipped. This observation is important for distributed embedded systems. For instance, the TelosB RF transceiver requires 23mA to operate the receiving mode whereas the processor just uses 1.8mA [MEM04]. In such a scenario, the run time improvements gets overshadowed by the longer and more expensive data transmission time.

Scenario	I-Frames	P-Frames	B-Frames
eledream 128	3	60	65
eledream 128 (50% perforated)	3	65	60
eledream 128 (75% perforated)	3	82	43

Table 6.5: Frame types of encoded x264 videos.

Finally, we emphasize that QCAPES compares the output of a reference implementation with the output of the approximated version. Therefore, in the case of the approximated video encoder, it is possible to produce better approximated output compared to the reference implementation if all compression steps are skipped. This explains the rather good output quality of the approximated video encoder in this case study. QCAPES follows this method, since it is not always possible to obtain access to the source material. Furthermore, the framework analyses the impact of the approximation and in the case of the video encoder, not the performance of the compression.

6.5.2 Approximated Image Compression

In the approximated image compression case study, the goal is to evaluate QCAPES capabilties to analyze the impact of algorithmic choice and reduced data type precision. We choose the `cjpeg` encoder from the jpeg-9b package [Ind18] for this use case. The encoder performs a JPEG compression and provides three different DCT algorithm (*float*, *int*, *fast*) implementations. Furthermore, the application provides an adjustable compression quality.

Table 6.6 lists the test input pictures for the compression algorithm. The test set comprises three standard test pictures (512×512 pixel) [Ima18], and a large picture (4000×4500 pixel) [Xip18]. In addition, test images from the PAMONO data set [SZS14]

were used. In a preprocessing step, the pictures were manually converted into the ppm-format or pgm-format expected by `cjpeg` as input. In the evaluation, we analyzed the impact of different DCT algorithms on the output quality, execution time and energy consumption. Further, the effect of different compression quality levels was analyzed.

Image	Depth	Dimension	Size
bigbuckbunny (bbb)	8-bit	4000×4500	52 MB
lena_color	8-bit	512×512	769 kB
lena_grey	8-bit	512×512	257 kB
monkey_color	8-bit	512×512	759 kB
pamono	8-bit	1024×350	701 kB

Table 6.6: Input data for approximated image compression case study.

Table 6.7 shows the results of the SFM analysis for the `lena_color` test input. The DCT *float* algorithm is used as the reference for the quality assessment. The results confirm that with reduced target compression quality the overall output quality reduces as well. In addition, the *int* DCT version produces according to the SFM values the better images. Table 6.8 and Table 6.9 show the result of the PQVM evaluation. QCAPES multi-metric evaluation revealed that the MAE decreases with lower quality settings, while the MSE and RMSE increase. A possible explanation is that small errors vanish, while larger errors increase. The `cjpeg` documentation supports this hypothesis. It says that the fast implementation tends to produce worse results on higher quality settings than on lower settings. This shows that it is always advisable and recommended not to rely on just one specific metric or metric family, but to involve multiple metrics in the evaluation. Similar results were obtained for the other inputs. Therefore, we conclude, that all DCT algorithm implementations and the different data types achieve a comparable output quality. However, do the different algorithms provide an additional benefit in file size or run time and energy consumption?

DCT	Quality	MSE	RMSE	MAE	PSNR
int	100	0.12	0.34	0.12	57.48
int	80	0.24	0.49	0.15	54.40
int	50	0.27	0.52	0.08	53.76
int	30	0.22	0.47	0.06	54.69
int	10	0.28	0.53	0.02	53.67
fast	100	0.72	0.85	0.56	49.55
fast	80	1.05	1.02	0.55	47.92
fast	50	1.09	1.04	0.40	47.75
fast	30	1.21	1.10	0.33	47.32
fast	10	1.48	1.22	0.18	46.41

Table 6.7: Impact of quality settings for lena_color on the luminescence component.

Table 6.10 lists the resulting file sizes. As expected, the file size decreases with the target quality. However, in this case study, the deviation in file size is small compared to the approximated video encoder use case. Finally, Figure 6.7 shows the run time and

DCT	Quality	4x4	4x4b	8x8	8x8b	16x16	16x16b
int	100	0.99428	0.99419	0.99672	0.99663	0.99828	0.99823
int	80	0.98791	0.98727	0.99359	0.99345	0.99695	0.99671
int	50	0.99239	0.99325	0.99527	0.99668	0.99777	0.99792
int	30	0.99547	0.99664	0.99682	0.99861	0.99814	0.99856
int	10	0.99839	0.99849	0.99864	0.99945	0.99892	0.99907
fast	100	0.97488	0.97631	0.98529	0.98512	0.99229	0.99218
fast	80	0.95390	0.95303	0.97649	0.97502	0.98901	0.98873
fast	50	0.95693	0.96328	0.97195	0.98091	0.98735	0.98671
fast	30	0.97205	0.97994	0.97948	0.99029	0.98880	0.99687
fast	10	0.98721	0.99245	0.98806	0.99592	0.99192	0.99399

Table 6.8: UIQI values for Y-component of lena_color with different window settings.

DCT	Quality	4x4	4x4b	8x8	8x8b	16x16	16x16b	filter
int	100	0.99894	0.99893	0.99918	0.99916	0.99944	0.99943	0.99910
int	80	0.99835	0.99835	0.99874	0.99877	0.99915	0.99913	0.99859
int	50	0.99860	0.99870	0.99890	0.99896	0.99927	0.99936	0.99877
int	30	0.99899	0.99913	0.99913	0.99926	0.99938	0.99939	0.99908
int	10	0.99923	0.99939	0.99932	0.99952	0.99947	0.99954	0.99929
fast	100	0.99478	0.99517	0.99600	0.99601	0.99730	0.99729	0.99558
fast	80	0.99350	0.99379	0.99504	0.99506	0.99667	0.99664	0.99449
fast	50	0.99376	0.99444	0.99473	0.99452	0.99646	0.99652	0.99440
fast	30	0.99437	0.99538	0.99519	0.99515	0.99676	0.99682	0.99497
fast	10	0.99461	0.99650	0.99452	0.99439	0.99607	0.99618	0.99464

Table 6.9: SSIM values for Y-component of lena_color with different window settings.

energy consumption for processing the large bigbuckbunny (bbb) image. The results are similar, thus using different algorithms and data types in this use case study did not provide large improvements.

	Quality				
DCT	100	80	50	30	10
float	229.053 kB	44.013 kB	24.288 kB	17.629 kB	9.683 kB
int	230.211 kB (+0.5%)	44.200 kB (+0.4%)	24.340 kB (+0.2%)	17.656 kB (+0.2%)	9.692 kB (+0.15%)
fast	232.047 kB (+1.3%)	44.175 kB (+0.3%)	24.316 kB (+0.1%)	17.645 kB (+0.09%)	9.683 kB (+0%)

Table 6.10: File size for approximated image compression case study for lena_color.

6.5.3 Discussion

The case studies demonstrated the capabilities of QCAPES and highlight the importance of a good metric and objective selection. Furthermore, these experiments emphasize the usefulness of an automatic, multi-metric assessment framework to discover previously unexpected effects of the naïve application of approximation techniques. The studies revealed that in cases where approximation techniques interfere with application-specific inherent optimizations, e.g., the application of psychoacoustic or visual models for lossy data compression, the consideration of additional metrics, such as file size (and, consequently, energy consumption for transfer or storage) need to be considered to efficiently apply approximation techniques without introducing undesirable side effects.

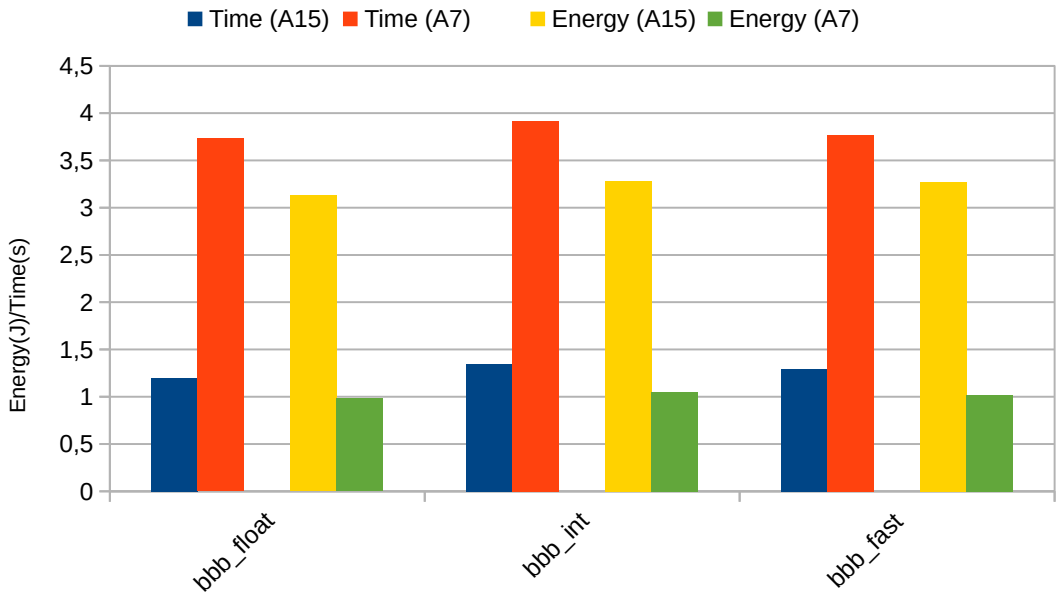


Figure 6.7: Run time and energy consumption for the cjpeg use case using different DCT implementations for compression of the big buck bunny image measured on an Odroid-XU3.

To conclude, we expect similar effects to show up especially in applications which are supposed to benefit most from approximation, thus multi-criterial assessment will be required for holistically building of efficient approximate systems.

6.6 Approximation in PICO - ApproxPICO

The case studies as well as the PAMONO virus detection optimization, presented in Chapter 5, show the potential of approximate computing. Major run time and energy consumption improvements can be achieved by leveraging the requirements on the output quality. However, implementing approximations safely is a complex and time-consuming task. With QCAPES, the PA4RES framework has access to a multi-metric assessment for the output quality. Consequently, a high-level specification of code regions which are sensible for approximation in combination with an automatic source-to-source transformations, transparent to the user, is a worthwhile goal. In the best case, a holistic approach should explore several approximation techniques automatically to find the optimal configuration for a given application targeting a specific hardware platform.

With ApproxPICO, we introduce the fundamentals for approximate computing into PA4RES. This enhancement opens the way for an automatic approximation during the parallelization process. Therefore, we extended PICO and the supported pragma set to annotate parallel regions which might benefit from approximation. The user annotates such parallel region with the `approximate` clause. This clause can either be added to parallel loops globally or pipeline stages locally. In both cases, the user must provide a

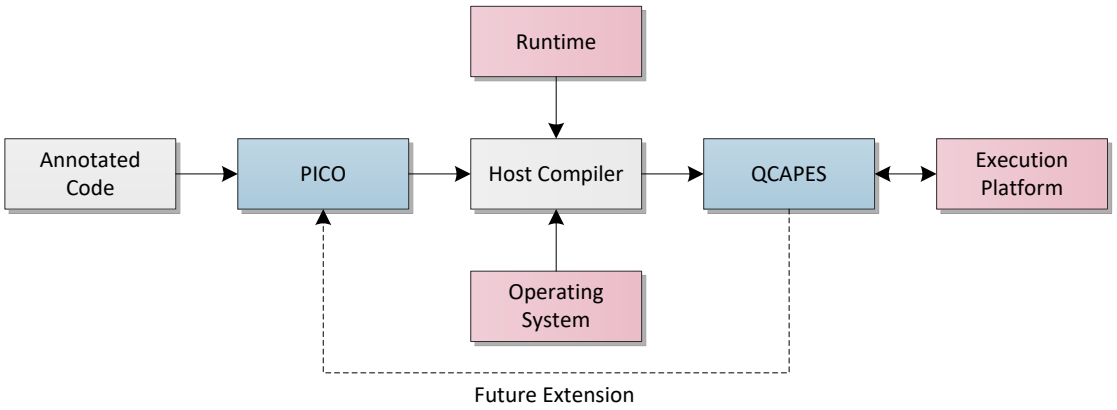


Figure 6.8: Automatic PICO approximation flow with QCAPES backend.

perforation rate such that PICO can employ *loop perforation*. A high perforation rate might resolve in the fact that some tasks are discarded. Tasks skipping was also proposed by [Rin07; Rin06]. PICO's data and control flow analysis keeps track of the perforation and takes care for the possible changed data dependencies during the parallelization process. Listing 6.1 shows exemplarily how to apply *loop perforation* to the first pipeline stage of the parallel pipeline. The `perfrate` clause controls the perforation and describes which iterations this section should process. Consequently, a `perfrate` of one means no approximation, a rate of two that every other iteration should be skipped. In case of a hybrid pipeline stage, the perforation rate must be specified for each parallel task similar to the `chunks` clause (cf. Subsection 3.5.1).

```
#pragma pico parallel pipeline for num_threads(2)
for (int i = 0; i < 10; ++i) {

  #pragma pico section taskid=1 approximate perfrate=2
  {
    // for loop ...
  }

  #pragma pico section taskid=2
  {
    // for loop
  }
}
```

Listing 6.1: ApproxPICO: Exemplary section perforation. Every other iteration is skipped for the first section.

Figure 6.8 visualizes the currently available tool flow provided by PA4RES to apply approximate computing techniques with PICO to sequential applications. The user provides the annotated application code to the framework. PICO parses the code and performs the instructed parallelization and applies the approximation. The modified parallel code and the reference version are then passed to the host compiler. QCAPES

	Base	Waiting	Read random	Read last
Execution time (s)	0.446	7.438	3.570	3.559
Energy consumption (J)	0.868	30.846	14.482	14.927

Table 6.11: Performance results for approximate communication experiment.

then performs a multi-metric evaluation on the target platform and the results are reported to the user. This manual approximation process offers some potential for future extensions.

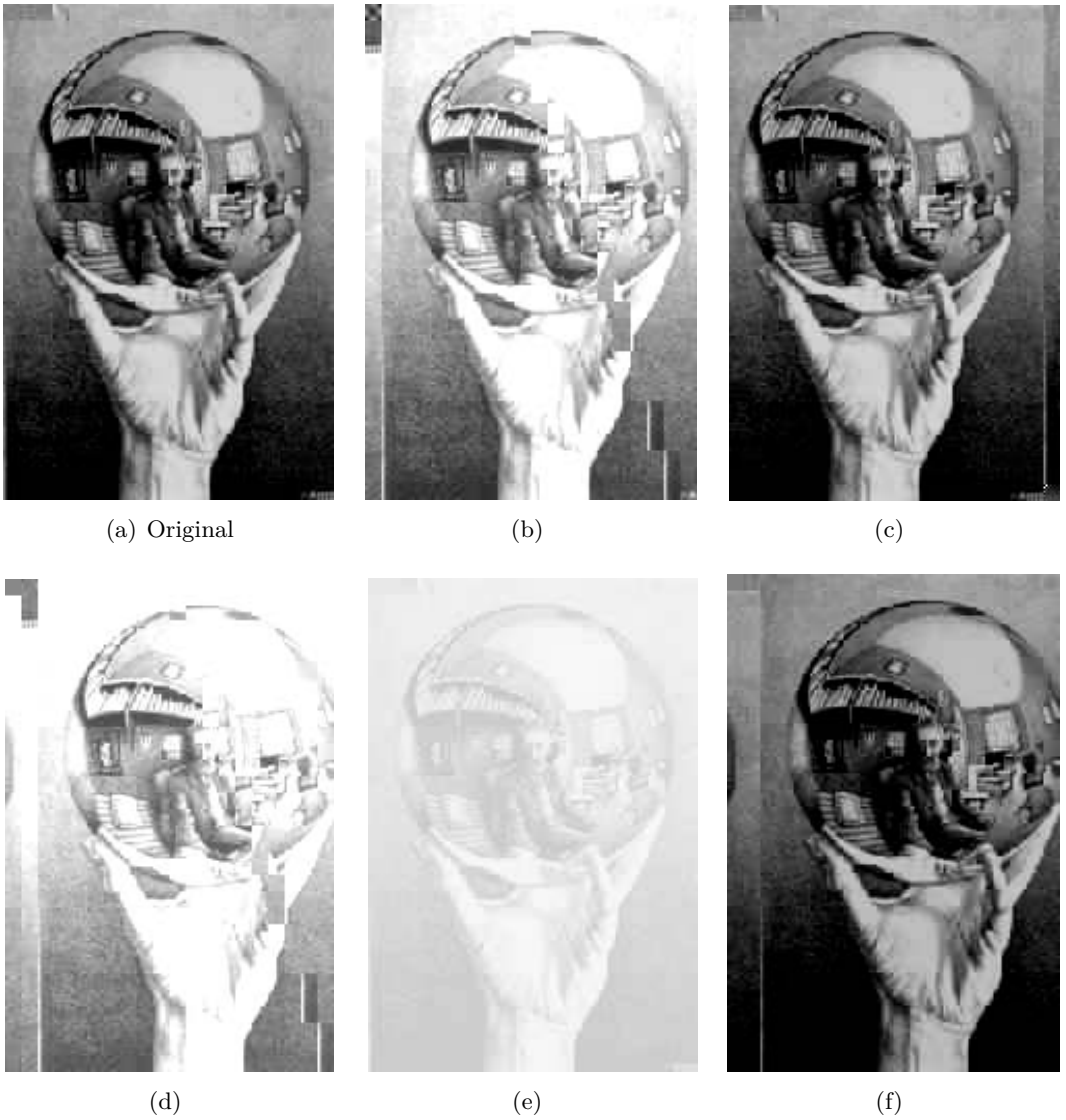
6.6.1 Approximate Communication - Case Study

Parallelization is limited by synchronization. Especially for future many-core systems with dozens of processors, this will be a serious concern. The fact exaggerates in cases where multiple parallel applications are executed simultaneously on the same system. Relaxed synchronization [RSN12] provides an interesting approach to leverage the waiting time on synchronization barriers. However, we propose to allow approximation during the data exchange [NEM15a]. In case of the FIFO-based data synchronization used by PICO, this approach applies approximation in case of a blocking FIFO access. In such an event, we propose several strategies. The first strategy re-uses the previously transmitted data, therefore, the FIFO semantic must be enhanced to deal with these circumstances. The second approach would use a predefined value and the last strategy would generate the data randomly.

In general, we expect that the approximate communication proposal can either be applied statically or during run time dynamically. In the static case, the user or PICO adds a waiting threshold to the approximated FIFO channels. Once that limit is reached, the data synchronization is approximated. Therefore, this approach adapts to a certain extent to the current workload during run time. The dynamic approach would extend QCAPES towards a runtime quality assessment to monitor the current output quality. If the quality is still good enough, some data synchronization points might be relaxed.

To evaluate the potential of such an approximate communication technique, we added new directives to ApproxPICO. The `wait(time)` clause instructs ApproxPICO to use approximate polling-based FIFOs with a given waiting threshold. With the clause `reuse`, ApproxPICO reused the previous data, otherwise it will generate random data.

We selected the `JPEG` benchmark and the Odroid-XU3 platform to evaluate the capabilities offered by approximate communication. For this preliminary case study, we introduced (random) artificial delays between 1 and 1000 μs in the FIFO read function and set the threshold to 100 μs. We conducted 100 repetitions for each setting and averaged the values. Table 6.11 shows the resulting execution times and energy consumption for the different strategies. As expected, we observe an improvement in run time and consequently energy consumption with approximate communication. However, the resulting images show some major distortions (cf. Figure 6.9). In some cases the errors are acceptable, but in others, we observed a shift in the entire picture. Therefore,



(a) Original (b) (c)

(d) (e) (f)

Figure 6.9: Impacts of approximate communication.

we conclude, that a deeper analysis is necessary to understand the side effects and benefits of approximate communication. In addition, an automatic exploration of approximate communication seems promising.

6.7 Conclusion and Future Work

The performance and energy walls expose a serious challenge for system designers and software developers. Traditional techniques like higher frequencies or multi-core designs are unable to break these walls. Leveraging the requirements onto the output quality to gain run time or energy improvements might be able to overcome these walls. At least, shifting these walls might provide additional time to develop new techniques. Approximate computing is a promising approach especially in the resource-restricted mobile systems domain which faces new challenges due to increased performance requirements. However,

an estimation of the impact on the final result of specific approximation techniques is still unclear. Several quality metrics can be used to analyze the results. Finding the correct metric in a set of dozen metrics which might be to some extent contradicting is a challenging, time-consuming and error-prone task.

The Quality Comparison for Approximate Programs on Embedded Systems (QCAPES) assessment framework provides a multi-metric evaluation and supports the user to introduce approximation into existing application safely. Furthermore, this framework enables an early analysis of the impact on the output quality and other performance characteristics during the development of new approximation techniques. With two case studies, we demonstrate QCAPES capabilities and the benefits of a mutli-metric evaluation. Although the results show that approximate computing can lead to an increase in performance, they also revealed that a careless usage of approximation techniques may interfere with application-specific inherent optimizations.

With ApproxPICO, we provide the fundamental work to consider approximation during the parallelization of sequential applications within PA4RES. Based on PICO's control and data flow analysis, ApproxPICO provides a simple way to annotate parallel regions which may benefit from approximation. ApproxPICO then automatically applies *loop perforation* to the annotated regions and takes care of changed dependencies, task skipping and necessary data synchronization. Parallel application might suffer from blocking data synchronization. With a preliminary case study, we investigated the opportunities offered by approximate communication where the application proceeds in the case of blocking FIFO read with random or previously received data. However, this promising approach offers several directions for future extensions.

Future Work The impact of approximations on other application from various domains may reveal additional interesting insights. Furthermore, it might be worthwhile to investigate the side effects of other commonly used approximation techniques. However, in the following, we present possible future extensions to ApproxPICO and PA4RES towards an automatic approximation exploration framework.

Feedback Loop: In the current state, the approximation flow provided by ApproxPICO and PA4RES does not include a feedback loop. Figure 6.8 indicates how such a feedback loop might interact with the processing flow. Results are then feed back to PICO which then integrates the new knowledge into the transformation process.

Additional Approximation Techniques: The current framework only provides *loop perforation* and consequently task skipping. Therefore, the integration of additional approximation methods into the framework might be useful. Many iterative algorithms can be approximated using an early loop termination. In such a case, the user could provide a threshold quality and PICO calculates the iteration range of that computation loop statically. Another promising technique is the support for algorithmic choice, here, the user would provide several implementations of the same algorithm. With special

annotations, the user would made PICO aware of these choices. Finally, PICO's tight integration with the ICD-C compiler framework allows an easy exchange of data types with a reduced precision. This technique is commonly used to map floating-point numbers to fix-point or even integers in order to improve the overall performance. With Florian Schmoll's PropCC approach that provides a deep analysis of reliable and unreliable data, we already have the fundamentals available to determine the impact of such data approximation statically.

Approximate Communication: The preliminary case study showed that, approximate communication can improve the performance of an application suffering from long blocking times. However, the impact on the output quality must be analyzed in more detail. In this case study, the approximation was manually applied to all FIFO channels, a more sophisticated selection might improve the output quality. In general, which strategy should be applied in which case is still unclear and offers interesting research opportunities. An automatic exploration of approximate communication done by ApproxPICO with a detailed mutli-metric assessment is a worthwhile goal.

Automatic Approximation Exploration: Finding a good trade-off between the available parallelization and approximation to achieve a certain quality and performance is a complex and time-consuming task. If this is done improperly, the benefits gained due to approximation might by overshadowed by drawbacks exposed by other objectives. Therefore, we suppose that an automatic approximation exploration considering the limitation of embedded systems and the target application is a worthwhile goal.

Chapter 7

Conclusion and Future Work

Contents

The quest for more performance of applications and systems became more challenging in the recent years. Especially in the cyber-physical and mobile domain, the performance requirements increased significantly. Former high-performance applications emerge in the area of resource-constrained mobile systems. Modern heterogeneous high-performance Multiprocessor System-on-a-Chip (MPSoC) provide a solid foundation to satisfy the high demand. Such systems combine general processors with specialized accelerators. Today's systems provide accelerators ranging from Graphics Processing Units (GPUs) to chips focusing on machine learning applications.

On the other side of the performance spectrum, the demand for small energy efficient systems exposed by modern Internet of Things (IoT) applications increased vastly. Such battery-driven systems may operate on remote places to take samples. In extreme cases, these systems are charged by unsteady power sources like solar panels. Others may be integrated in machines or smart homes. Several predictions say that these class of systems will easily outnumber the number of today's existing computers.

Developing efficient software for such resource-constrained multi-core systems is an error-prone, time-consuming and challenging task. To reduce the time-to-market, the developers must adapt existing sequential software to these new systems. To further improve the execution time or energy consumption, the developer might leverage the requirements on the output quality. Developers are forced to find good configurations with respect to execution time, energy and memory consumption and output quality. This thesis provides with PA4RES a holistic semiautomatic approach to parallelize and implement applications for such platforms efficiently. Our solution supports the developer to find good trade-offs to tackle the requirements exposed by modern applications and systems.

7.1 Summary of Contributions

The initial challenges discussed in Chapter 1 revealed many aspects today's developer must take into account to utilize the capabilities of modern embedded MPSoCs. In this thesis, we propose several solutions to tackle the exposed challenges. Our approaches consider energy consumption, execution time, memory consumption and output quality. In the following, we provide a comprehensive summary of the goals which were accomplished.

PA4RES: The Parallelization for (4) Resource-restricted Embedded Systems (PA4RES) methodology provides a holistic approach to parallelize sequential application with respect to several objectives. The framework provides a seamless interaction of various tools. Besides the approaches developed in this thesis, it contains the Parallelism Extraction for Embedded Systems (PAXES) parallelizer, a Performance Estimator and several other useful tools. PA4RES abstracts from concrete target platform with an internal high-level system model. However, in this thesis, we present several backends to generate code for simulators and real hardware. Furthermore, we developed two energy and performance measurement applications for the real hardware platform.

PICO: The Parallelism Implementer and Communication Optimizer (PICO) approach enables developer to express complex parallelism with simple, high-level directives added to the sequential code. Following the PA4RES philosophy, our approach preserves the sequential source code which improves the understandability and maintainability of parallel software. The developer can express complex parallelism, like hybrid pipeline parallelism, with only a few annotations to the code. In combination with the ICD-C compiler framework, PICO automatically takes care of necessary data synchronization and implements required data exchanges. Therefore, it performs several data flow analysis to detect data dependencies. In addition, PICO offers manual and automatic load balancing for heterogeneous architectures.

Parallel software often suffer from data synchronization, therefore, an efficient data exchange is essential to achieve the required performance. PICO provides a sophisticated approach to explore a large communication-related parameter space. The evolutionary optimization algoritm provisions the FIFO channel size, the mapping, selects implementation details and merges channels to reduce overhead. During this process, PICO considers execution time, energy consumption and memory footprint. We demonstrated the importance of our approach with an extensive evaluation where we found solutions with a speedup of 1.8 or a reduction in energy consumption of 30%. Using a high-level execution model we could reduce the overall optimization time significantly by still achieving useful results.

PAMONO Virus Detection: Traditionally, biological virus detection was a time-consuming task and usually bound to large laboratories. The Plasmon-Assisted Microscopy of Nano-Objects (PAMONO) sensor approach enables a fast virus detection

using optical methods. With a sophisticated virus detection software, a real-time virus detection was achieved. Within this thesis, we accomplished a mobile virus detection solution. Using a Design Space Exploration (DSE) algorithm exploring various software- and hardware-related parameters we were able to derive a soft real-time capable virus detection running on a high-performance embedded system, commonly found in today's smart phones. Compared to a baseline implementation, our solution achieved a speedup of 4.1 and 87% energy savings satisfying the soft real-time requirements. Accepting a degradation of the detection quality, which still is usable in a medical context, led to a speedup of 11.1. This work provides the fundamentals for a truly mobile real-time virus detection solution.

Approximate Computing and ApproxPICO: Today's system designers and software developers face serious challenges exposed by the performance and power walls. Traditional techniques to tackle these walls fail. Approximate computing is a promising approach to overcome or at least shift these walls. By accepting a degradation in the output quality, developer can achieve additional improvements in terms of execution time or energy consumption. Finding a good trade-off is a complex task. Various existing quality metrics might predict a contradicting impact on the output quality. Especially for a safe integration of approximation into existing application or during the development of new approximation techniques, a method to assess the impact on the output quality is essential. With Quality Comparison for Approximate Programs on Embedded Systems (QCAPES), we provide a multi-metric assessment framework to analyze the impact of approximation. Qualitative case studies demonstrated the usefulness of our approach and reveled side effects of a careless usage of approximation techniques.

With ApproxPICO we propose an extension to PICO to consider approximate computing during the parallelization of sequential applications. Embedded in the PA4RES framework, the tool applies a source-to-source approximation technique to perforate the iteration space of loops. Under certain conditions, this might lead to task skipping. The transformed application can automatically be assessed by QCAPES to get insights into the impact of the approximation. This work built the base for an automatic exploration of several approximation techniques. In addition, we propose approximate communication to improve the performance of applications suffering from long blocking times. Especially for future many-core systems with dozens of processors, this will be a serious concern. The fact exaggerates in cases where multiple parallel applications are executed simultaneously. In case of the FIFO-based data synchronization, this approach applies approximation in case of a blocking FIFO access. In such an event, we propose several strategies. The first strategy re-uses the previously transmitted data, therefore, the FIFO semantics was enhanced to deal with these circumstances. The second would use a predefined value and the last would generate the data randomly. With a preliminary case study, we investigated the opportunities offered by approximate communication. This promising approach offers several directions for future extensions.

7.2 Future Work

The goals accomplished within this thesis opened various directions for improvements and future work. In the following, we provide a comprehensive overview of possible future extensions and improvements.

PICO and PAXES: PICO demonstrated in this thesis its importance during the parallelization of sequential application and for the optimization of necessary data synchronization. However, there is some room for improvements. On the technical side, a more precise data analysis would reduce the risk of too conservative data synchronization. Furthermore, a more precise analysis might help to remove some restrictions exposed to the source code.

Efficient data synchronization is vital for the overall performance of parallel applications and PICO provides an approach to implement this efficiently. However, we identified several ways for a future extension of our approach. The GA generates new unknown solutions and evaluates them with a costly fitness evaluation. If such a solution that is worthwhile to evaluate is currently unknown and thus (costly) evaluated. Fortunately, model-based optimization (MBO) offers a method to predict if an individual solution should be evaluated to gain new knowledge or not. We used MBO in the context of the CRC and in [KLN17] to estimate run time and performance of specific machine learning algorithms on the Odroid-XU3 platform. Therefore, to prune the evaluation time of PICO's genetic algorithm, a combination with a MBO approach seems promising.

Overall, a holistic model-based approach, which does not require a simulation, seems to be a worthwhile goal. Therefore, we need to fine-tune the underlying data that the execution model uses to improve the estimations.

This thesis demonstrates the importance of data synchronization and its implementation. At the current state, PAXES assumes a static FIFO implementation and only uses a fairly abstract high-level communication cost model. Knowledge discovered by PICO regarding the actual synchronization impacts could be fed back into PAXES' parallelization algorithm. Thus, knowledge gathered by PICO could be used during the parallelization process to further improve the results. Finally, we would like analyze PICO's capabilities regarding other benchmark and new hardware platforms.

PAMONO Virus Detection: In the current state, the PAMONO sensor only allows to detect a stem of viruses which attach to the same type of antibody. To increase the detection range, a (larger) gold layer can be partitioned and coated with different antibodies. The detection process gets complicated with each additional virus type but we showed that some headroom is available. In this case, the image resolution must be adapted to the new gold layer size leading to a higher computational demand. Instead of leveraging the detection quality, the DSE framework could be extended to support independent optimization for separate regions in the recorded image. In such case, the optimization algorithm might increase the detection quality in some areas while reducing

the quality in others. Instead of detecting multiple virus types simultaneously, the available headroom can be used to increase the resolution of the images and still meet the real time and energy requirements. Finally, a cheaper hardware system might be sufficient to achieve the virus detection task.

The next step is to move towards an areal monitoring system with multiple sensors placed at public locations like airports. To achieve a certain cost-effectiveness, it might be necessary to split the data sampling and processing task. In such a case, inexpensive low-power embedded systems take the sample images and send them to a central powerful computer to evaluate the data streams.

Approximate Computing and ApproxPICO: The impact of approximations on other applications from various domains may reveal additional interesting insights. Furthermore, it might be worthwhile to investigate the side effects of other commonly used approximation techniques.

In the current state, the approximation flow provided by ApproxPICO and PA4RES does not include a feedback loop. We think that such a feedback loop enables an automatic approximation exploration considering the resource limitation of embedded systems.

ApproxPICO only provides loop perforation and consequently task skipping. Therefore, the integration of additional approximation methods into the framework might be beneficial. Many iterative algorithms can be approximated using an early loop termination. Another promising technique is the support for algorithmic choice. With special annotations, the user would made PICO aware of these choices. Finally, PICO's tight integration with the ICD-C compiler framework allows an easy exchange of data types with a reduced precision. This technique is commonly used to map floating-point numbers to fix-point or even integers in order to improve the overall performance.

Parallelization is limited by synchronization. The preliminary case study showed that, approximate communication can improve the performance of an application suffering from long blocking times. However, the impact on the output quality must be analyzed in more detail. In this case study, the approximation was manually applied to all FIFO channels, a more sophisticated selection might improve the output quality. This approach adapts to a certain extent to the current workload during run time. A dynamic approach would extend QCAPES towards a runtime quality assessment to monitor the current output quality. If the quality is still good enough, some data synchronization points might be relaxed. In general, which strategy should be applied in which case is still unclear and offers interesting research opportunities. An automatic exploration of approximate communication done by ApproxPICO with a detailed mutli-metric assessment is a worthwhile goal.

Appendix A

Appendix

Contents

A.1 EnergyMetric - CoMET-Systems

Energy measurement for a simulator-based platform (cf. Section 2.1) requires an energy model. The Virtualizer CoMET simulator platform provides a method for user-defined metric modules. These modules enable monitoring and manipulation of the simulator, e.g. for debugging. We implemented an energy metric module named EnergyMetric to model the energy consumption of our platforms. Florian Schmoll and the author of this thesis developed in collaboration the EnergyMetric module. Other theses [Cor13; Hol17] used EnergyMetric as well. In general, the EnergyMetric module tracks important event of the processors, memories and interconnects. Each event is associated with an energy budget and accumulated to the total energy consumption. The measurement can be started and stopped with issuing a read of special memory locations. Applications can control the measurement through API calls.

The energy model and access times for the memories rely on data derived from CACTI [MBJ09], a state-of-the-art integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. CACTI has been verified with real memories. We modeled on/off chip memory both as DRAM and scratch memory. EnergyMetric therefore uses the corresponding values for the different memories in our systems. Stefan Steinke developed at our department an energy model for ARM7TDMI processors [SKW01]. The

	Frequency		
State	100 MHz	250 MHz	500 MHz
Active Cycle	917.007 fJ	918.127 fJ	921.729 fJ
Stall Cycle	605.340 fJ	606.460 fJ	610.063 fJ
Idle Cycle	92.007 fJ	93.127 fJ	96.729 fJ

Table A.1: Frequency-dependent high-level processor energy model.

simulation-based platforms used in this thesis uses ARM1176 processors, thus we adjusted the energy values by comparing both processor capabilities as well as trends in structure size and overall energy consumption. Since we do not compare different platforms itself, we argue that these energy values allow a comparison of the performance of different application implementations on the same platform. Currently, the EnergyMetric accumulates active, stall and idle cycles for the processors. EnergyMetric uses frequency dependent energy values for the heterogeneous platform. Table A.1 lists the energy values for all processor states and frequencies used in this thesis.

A.2 Energy measurement on the Odroid-System

During this thesis, we developed two energy measurement applications for the Odroid-XU3 platform. The platform uses four INA 231 [Tex13] to sense voltage, current and power. A standardized I^2C interface connects the sensors are with the processor. The sensor provides configurable averaging and conversion times. The platform ships with predefined and fixed parameters in the original driver and Linux implementation. We extended the driver provided by the manufacturer to enable a configuration of the sensors dynamically during runtime. This enables a rapid analysis with different settings without rebooting the target platform. Our driver extensions are available at [SFB17b]. We developed two applications, the EnergyMeter and the Energy Relay Reader to make the measured sensor values easily accessible. In both applications, the systems measures the energy consumption in situ and stores the results time coded in a CSV file. This enables a straightforward graph generation e.g. with gnuplot. The measurement applications use standardized methods to access the sensor, thus, we think that our applications are usable for other sensors.

A.2.1 EnergyMeter

We derived the EnergyMeter implementation from the original energy measurement application provided by the manufacturer of the Odroid-XU3 platform. The original application provides a graphical user interface and the measurement could not be controlled externally. A graphical user interface does not fit into our remote evaluation concept. Thus, our implementation removes these limitations and extends the application to meet our requirements. The EnergyMeter uses polling to read out sensor values. Thus, it is able to sense dynamic changes with high resolution but produces therefore a high

CPU workload. External user-controlled events steer the sensing process. In this case, Linux named pipes trigger the measurement. This feature enables a measurement of specific parts of an application. We used EnergyMeter during the optimization of the PAMONO virus detection software [NLE15]. Here, we used this feature to exclude the initial setup phase of the virus detection.

A.2.2 Energy Relay Reader

A time precise readout of the sensor data is not always required, thus, we developed the Energy Relay Reader, which allows a relaxed, and CPU load friendly energy assessment. This application uses the relay feature of the Linux operating system. The basic idea is to create threads reading the energy sensor at specific points in time. These threads are then either scheduled to the worker or kernel thread queue. The queues differ in the priority they are processed. After the operating system processes the measurement elements in the queue, the application stores the results in files. This approach does not produce a high CPU load since the CPU could idle between the measurements. A drawback of this approach is the possibility that due to queue processing, some readouts get delayed drastically. It is also possible, that one measures longer as the runtime of the experiments. This happens if measurement tasks remain in the queues and are executed after the termination of the experiment. However, these drawbacks are acceptable for long running applications or average measurements.

A.3 PICO API and Runtime

This section presents a technical overview of the PICO API and runtime. All functions of the *pico.h* header file are listed and briefly described. A target runtime must implement all functions defined by the API. For some cases, we highlight differences between the RTEMS and POSIX implementation. The simulator-based platforms with the RTEMS operating system influenced the initial development of the PICO runtime strongly. Thus, for the POSIX implementation some methods are not required and therefore can be left empty. During this thesis, we ported the PICO runtime to the Odroid-XU3 system. The well-structured API allowed a fast implementation within hours.

```
void pico_init();
void pico_init_tasks(unsigned int numTasks);
void pico_init_barriers(unsigned int numBarriers);
void pico_init_semaphores(unsigned int numSemaphores);
void pico_init_channel_lookup(unsigned int numChannels);
```

Listing A.1: PICO API - Initialization Methods.

Listing A.1 lists the methods that the PICO runtime uses to initialize all necessary management data. It defines the number of tasks, initializes barrier and semaphores. Further, a lookup table for the used communication channels is created. These methods

are required if the underlying operating system, if any, does not provide built-in barriers, semaphores and task management.

```
void pico_register_task(void* functionptr, t_TaskID taskID);
void pico_create_task(t_TaskID taskID, t_ProcessorID processorID);
void pico_fork(t_TaskID taskID);
void pico_join(t_TaskID taskID);
```

Listing A.2: PICO API - Task Methods.

The RTEMS implementation used in this thesis uses a slightly different task handling. More details regarding this can be found in [Hei10]. In this implementation, tasks need to be created and registered before they can be executed. Listing A.3 shows PICO's task management interface functions. For a true POSIX compliant operating system, these methods just call their POSIX equivalent.

```
void pico_create_barrier(t_BarrierID barrierID);
void pico_wait_barrier(t_BarrierID barrierID);
void pico_create_semaphore(t_SemaphoreID semaphoreID);
void pico_wait_semaphore(t_SemaphoreID semaphoreID);
void pico_release_semaphore(t_SemaphoreID semaphoreID);
```

Listing A.3: PICO API - Synchronization Methods.

We defined the PICO runtime with platform independence in mind, thus, we provide a set of abstract synchronization primitives. Listing A.3 provides a basic set of methods to synchronize data access and execution. In the case that the target operating system supports native synchronization, the PICO function might use these methods.

```
void pico_init_channeltypes(unsigned int numChannels, t_ChannelType
    ↪  type);
void pico_init_channel(t_ChannelID channelID, t_ChannelType type,
    ↪   t_ChannelDevice device, unsigned int numelement,
    ↪ t_ElementSize elementsize);
void pico_init_channel_wait(t_ChannelID channelID, t_ChannelType
    ↪ type, t_ChannelDevice device, unsigned int numelement,
    ↪ t_ElementSize elementsize, unsigned int waitUS);
void pico_write_channel(t_ChannelID channelID, void* data);
void pico_read_channel(t_ChannelID channelID, void* data);
```

Listing A.4: PICO API - Communication Methods.

Beside task management and synchronization, the main job of the PICO runtime is to implement data exchange between parallel running tasks. Therefore, we provide a collection of API calls. Listing A.4 lists the communication channel functions. The basic idea of PICO is to decide between channel types and channel devices. A channel type might be an interrupt-based queue or a queue using busy waiting for blocking. A channel device describes the allocation to a communication infrastructure, for instance

a queue mapped to a SPM. The available devices and types are defined by the actual library implementation.

```
void pico_memcpy(void *dest, void *src, unsigned int size);
void* pico_malloc_dram(unsigned int size);
int pico_free_dram(void * mem);
void* pico_malloc_spm(unsigned int size);
int pico_free_spm(void * mem);
```

Listing A.5: PICO API - Miscellaneous Methods.

Some operating systems, like the employed RTEMS implementation, do not provide a dynamic memory (heap) management. Thus, PICO provides methods to either use a simple allocator, like buddy allocation, or use operating system methods. Listing A.5 lists the interfaces used to allocate data to the memory.

A.4 Execution Model - Performance Extraction

The execution model requires run time and energy data of the modeled FIFO implementation to calculate the estimated performance. Especially, the communication model needs a precise knowledge about what a successful FIFO and a blocking access costs. Therefore, we implemented simple parallel test cases which communicate data. We performed several measurements with different channel configurations with and without blocking. For instance, we varied the FIFO capacity and element size. We used as payloads for a channel 1 B,4 B,20 B,40 B,80 B,160 B,316 B,1024 B,4096 B and 16384 B. The FIFO capacity was 1, 10, 20 and 255.

The measured values were fed into a regression analysis to estimate the parameters of each implementation. The parameters *p2CyclesNumBytes*, *p1CyclesNumBytes*, *p2CyclesFIFOSize*, *p1CyclesFIFOSize*, *p3CyclesBoth* and *bCycles* were derived with R and led to the following model:

$$\begin{aligned} unifiedCycles = & p2CyclesNumBytes * numDataBytes * numDataBytes \\ & + p2CyclesFIFOSize * channelSize * channelSize \\ & + p1CyclesNumBytes * numDataBytes \\ & + p1CyclesFIFOSize * channelSize \\ & + p3CyclesBoth * (channelSize * numDataBytes) \\ & + bCycles \end{aligned}$$

	Execution time			Energy consumption		
# Bytes	Model	Simulation	Diff	Model	Simulation	Diff
1	13488	13411	0.57 %	1197.13	1142.97	4.52 %
4	13096	13536	-3.36 %	912.59	1157.05	-26.79 %
20	15153.5	14196	6.32 %	1415.13	1230.80	13.03 %
40	15739	15004	4.67 %	1477.92	1319.08	10.75 %
80	16808	16562	1.46 %	1480.05	1479.37	0.05 %
160	18554.5	19451	-4.83 %	1616.57	1711.93	-5.90 %
316	23531.5	24208	-2.87 %	1680.77	1686.52	-0.34 %

Table A.2: Read from a RTEMS-based FIFO with a capacity of 10.

	Execution time			Energy consumption		
# Bytes	Model	Simulation	Diff	Model	Simulation	Diff
1	4921	4897	-0.49 %	30.89	31.80	2.96 %
4	4921	4889	-0.65 %	30.89	32.08	3.87 %
20	4896	4854	-0.86 %	32.33	33.61	3.98 %
40	4894	4830	-1.31 %	35.55	35.62	0.18 %
80	4894	4842	-1.06 %	41.60	40.03	-3.78 %
160	4894	5115	4.52 %	54.25	51.27	-5.50 %
316	9894	6601	-33.28 %	141.70	88.00	-37.89 %
1024	34905	35512	1.74 %	592.09	544.53	-8.03 %
4096	124902	127116	1.77 %	2509.70	1999.78	-20.32 %
16384	519776	507177	-2.42 %	11219.46	11341.04	1.08 %

Table A.3: Read from an interrupt-based FIFO with a capacity of 1.

$$
\begin{aligned}
unifiedEnergy = & (p2EnergyNumBytes * numDataBytes * numDataBytes + \\
& + p2EnergyFIFOSize * channelSize * channelSize \\
& + p1EnergyNumBytes * numDataBytes \\
& + p1EnergyFIFOSize * channelSize \\
& + p3EnergyBoth * (channelSize * numDataBytes) \\
& + bEnergy) * unifiedCycles
\end{aligned}
$$

The unified cycles and energy values are used to model a successful, without blocking, FIFO access. In case of blocking the execution model uses averaged values of the measured blocking time for the interrupt and RTEMS channel implementation. In case of a sleep implementation, the model calculates the waiting time with respect to the configurable timer value. Tables A.2 and A.3 list additional results regarding the model and simulation comparison.

A.5 Digital and Physical Units

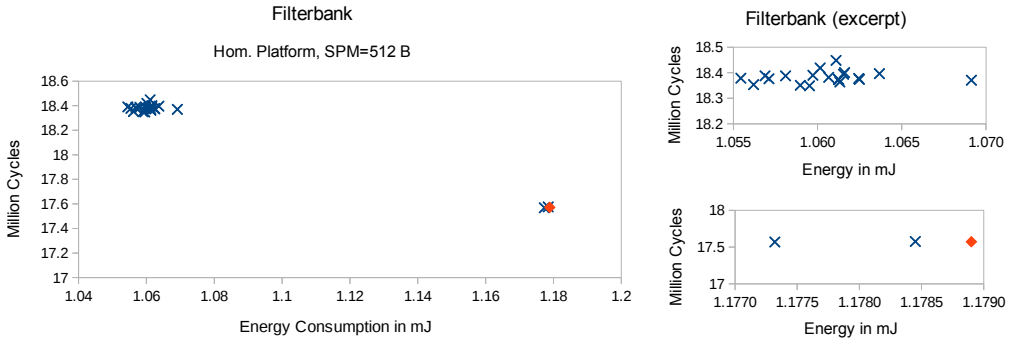
Within this thesis we use the following digital and physical units:

- Second: s
- Millisecond: ms
- Microsecond: μs
- Hour: h
- Joule: J
- Millijoule: mJ
- Femtojoule: fJ
- Byte: B
- Kilobyte: KB $\stackrel{\wedge}{=}$ 1024 B
- Megabyte: MB $\stackrel{\wedge}{=}$ 1024 KB
- Gigabyte: GB $\stackrel{\wedge}{=}$ 1024 MB

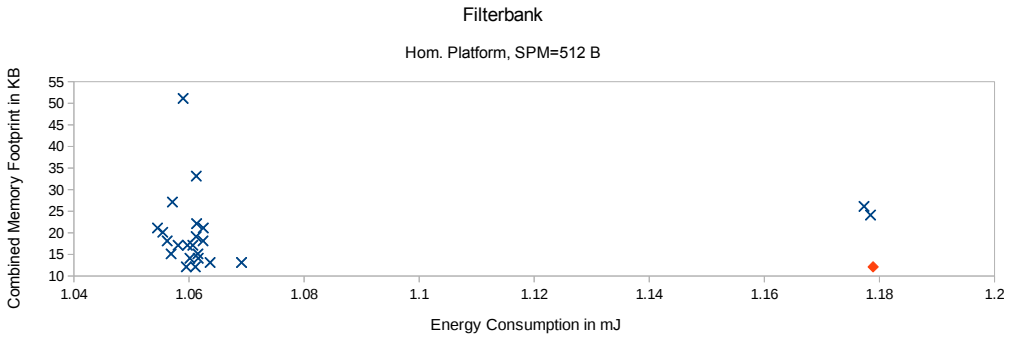
Appendix B

Results for Communication Optimization Experiments

In the following, we list the remaining results for the communication optimization experiments shown in Section 4.5. The order is: `Filterbank`, `Spectral`, `JPEG` and `PAMONO Preprocessing`.

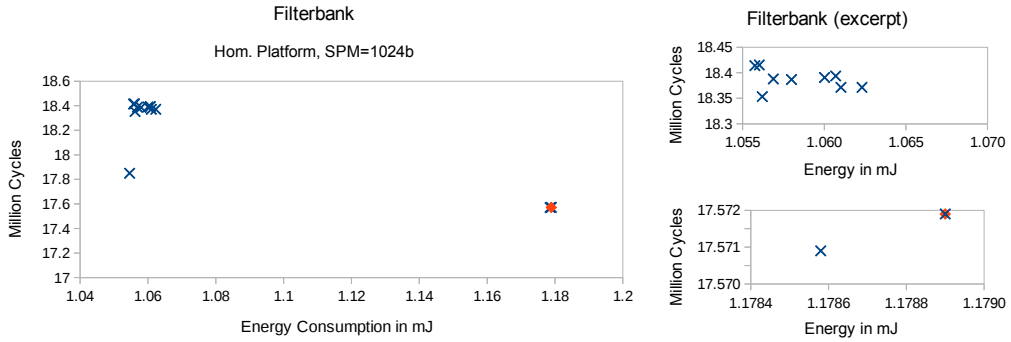


(a) Run time and energy consumption

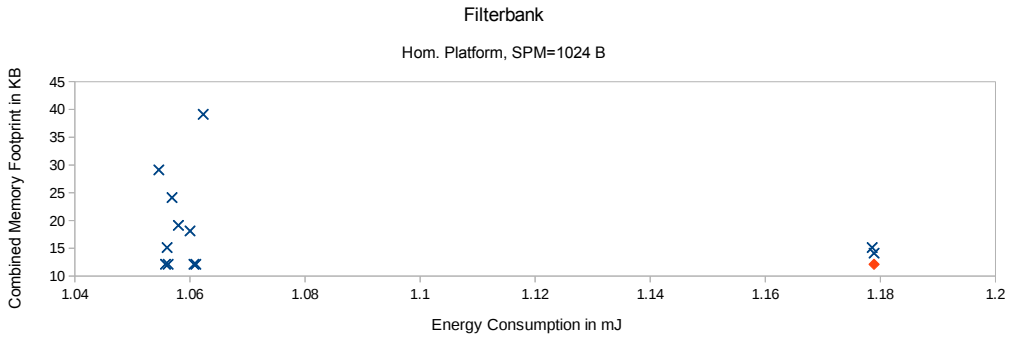


(b) Combined memory footprint and energy consumption

Figure B.1: Filterbank on the homogeneous system with SPM restricted to 512 B.

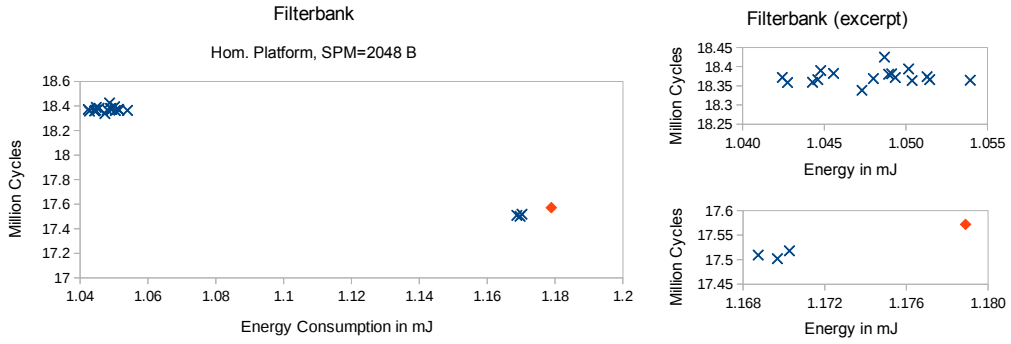


(a) Run time and energy consumption

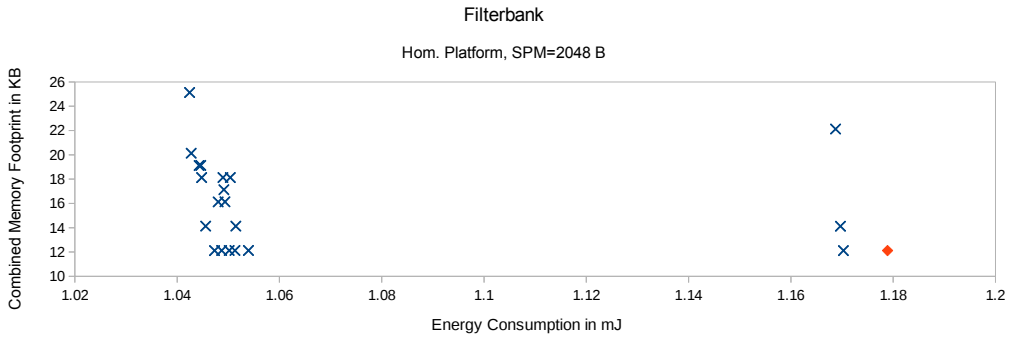


(b) Combined memory footprint and energy consumption

Figure B.2: Filterbank on the homogeneous system with SPM restricted to 1024 B.

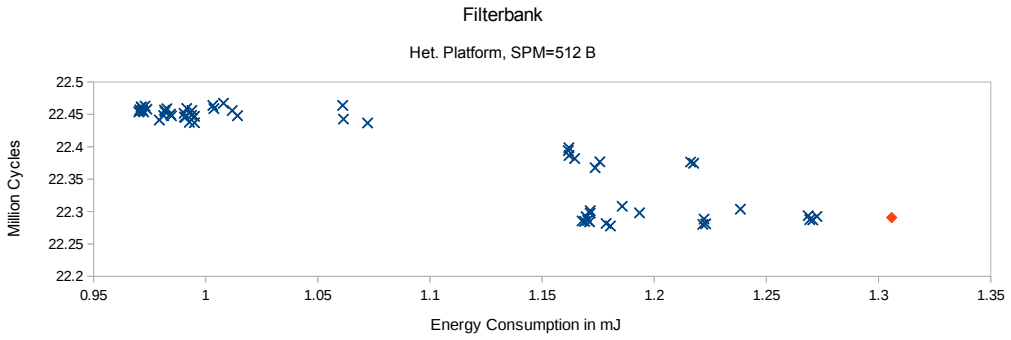


(a) Run time and energy consumption

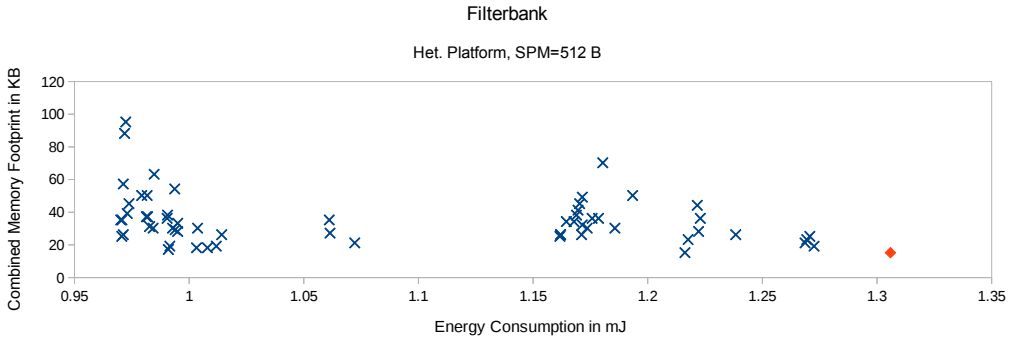


(b) Combined memory footprint and energy consumption

Figure B.3: Filterbank on the homogeneous system with SPM restricted to 2048 B.

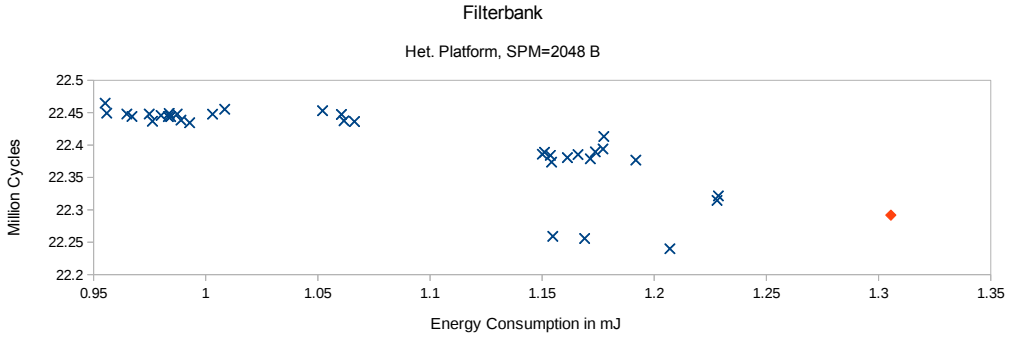


(a) Run time and energy consumption

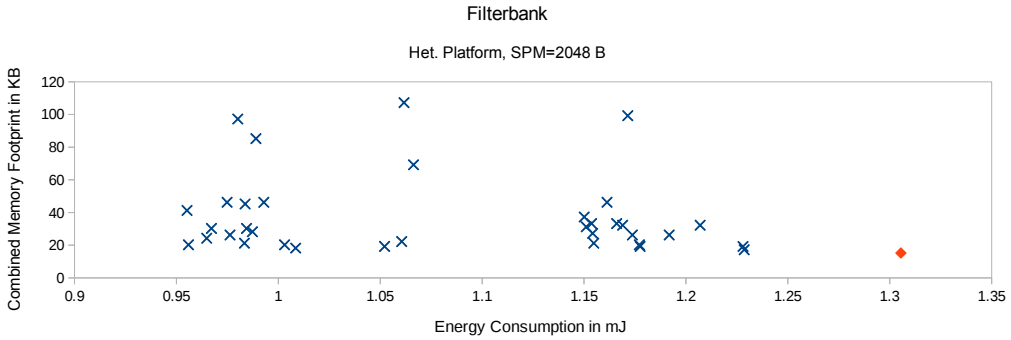


(b) Combined memory footprint and energy consumption

Figure B.4: Filterbank on the heterogeneous system with SPM restricted to 512 B.

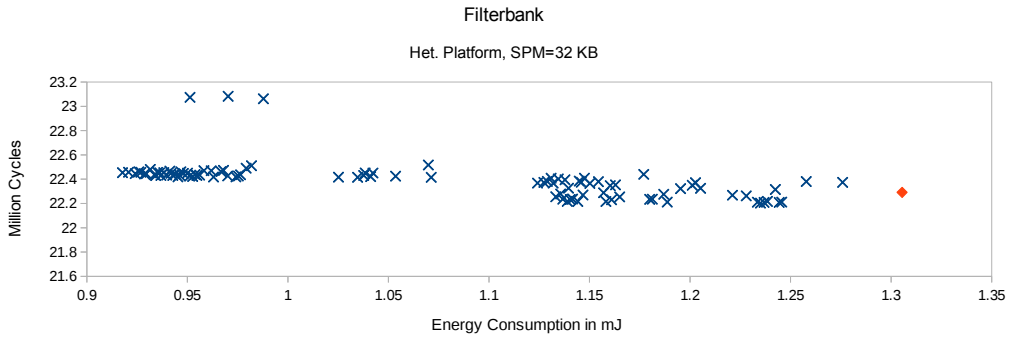


(a) Run time and energy consumption

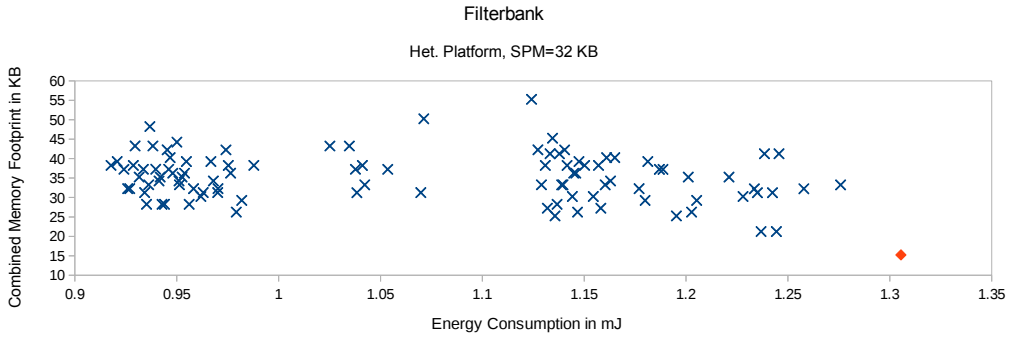


(b) Combined memory footprint and energy consumption

Figure B.5: Filterbank on the heterogeneous system with SPM restricted to 2048 B.

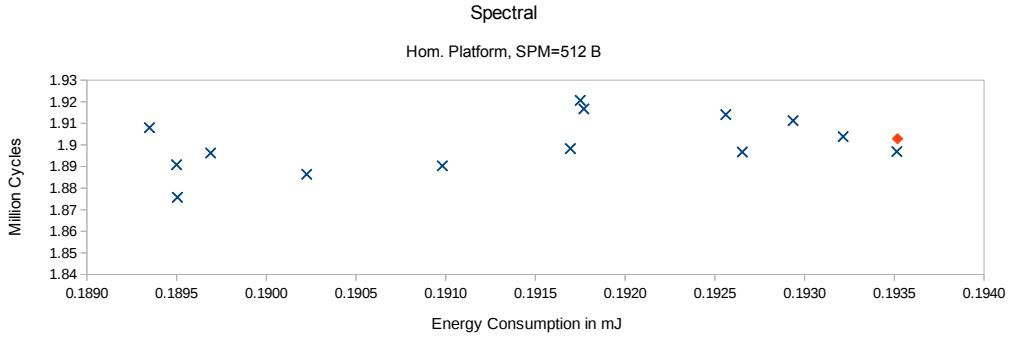


(a) Run time and energy consumption

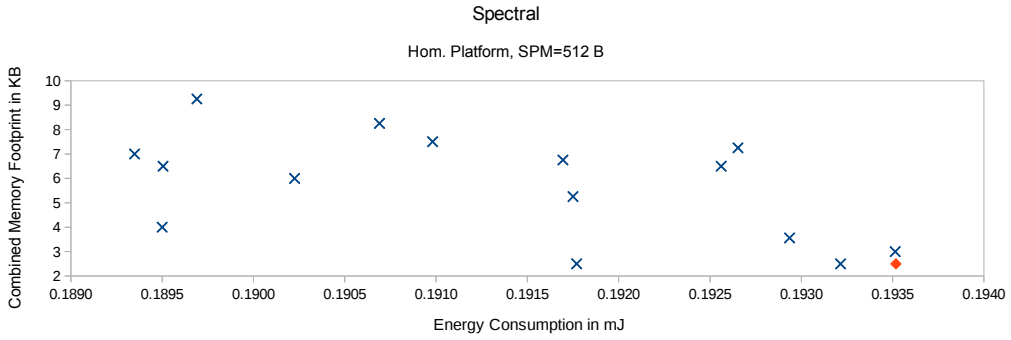


(b) Combined memory footprint and energy consumption

Figure B.6: Filterbank on the heterogeneous system with SPM restricted to 32 KB.

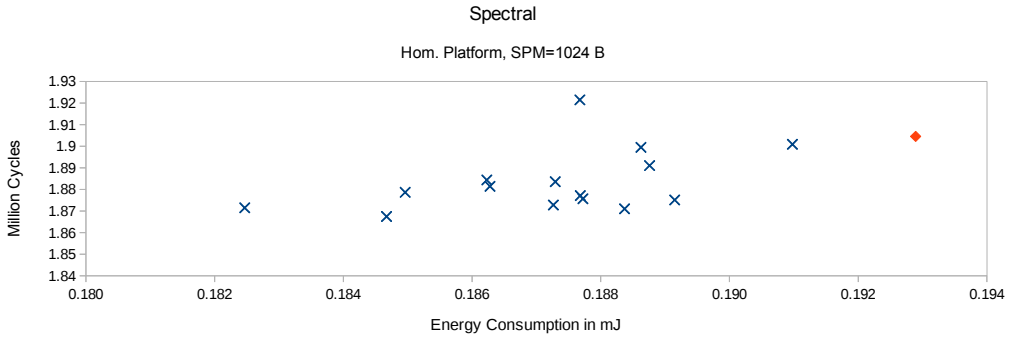


(a) Run time and energy consumption

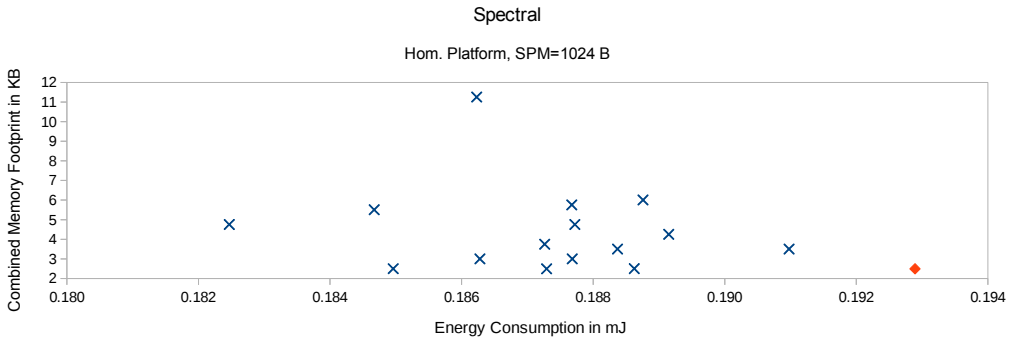


(b) Combined memory footprint and energy consumption

Figure B.7: Spectral on the homogeneous system with SPM restricted to 512 B.

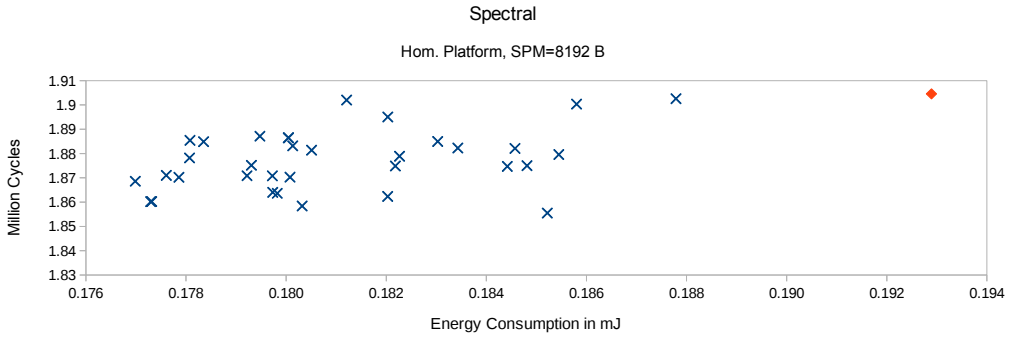


(a) Run time and energy consumption

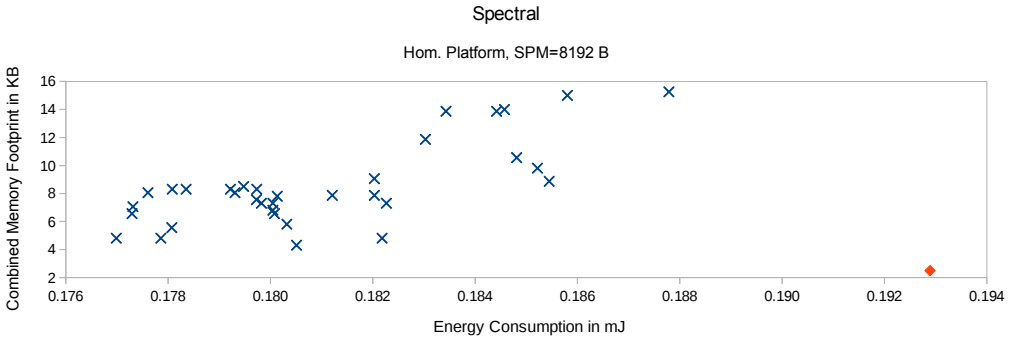


(b) Combined memory footprint and energy consumption

Figure B.8: Spectral on the homogeneous system with SPM restricted to 1024 B.

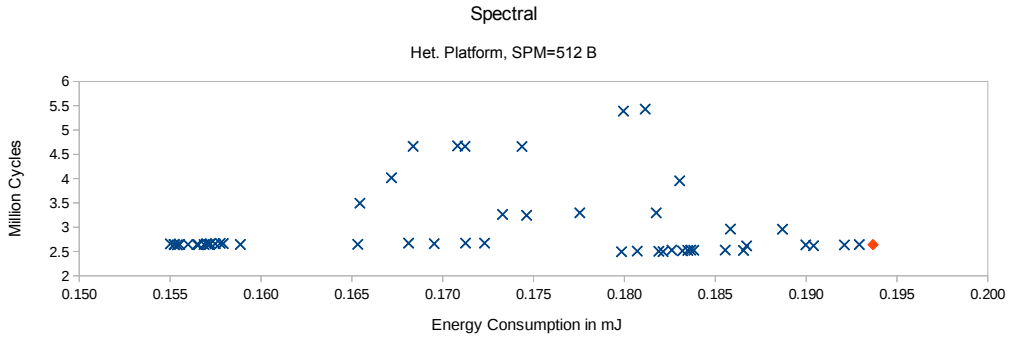


(a) Run time and energy consumption

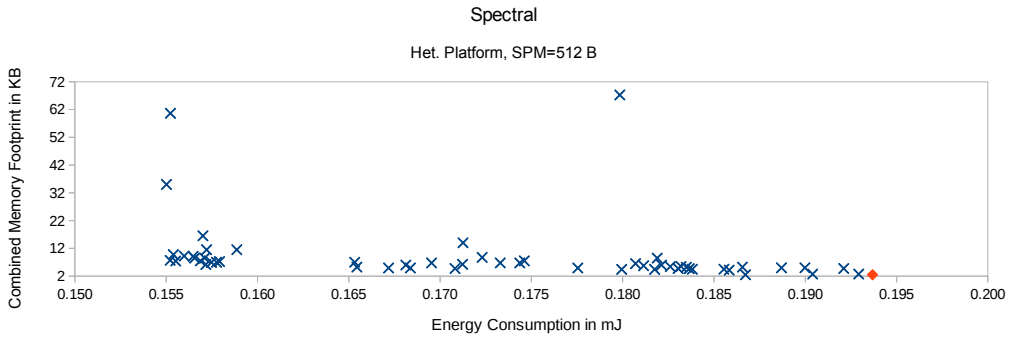


(b) Combined memory footprint and energy consumption

Figure B.9: Spectral on the homogeneous system with SPM restricted to 8 KB.

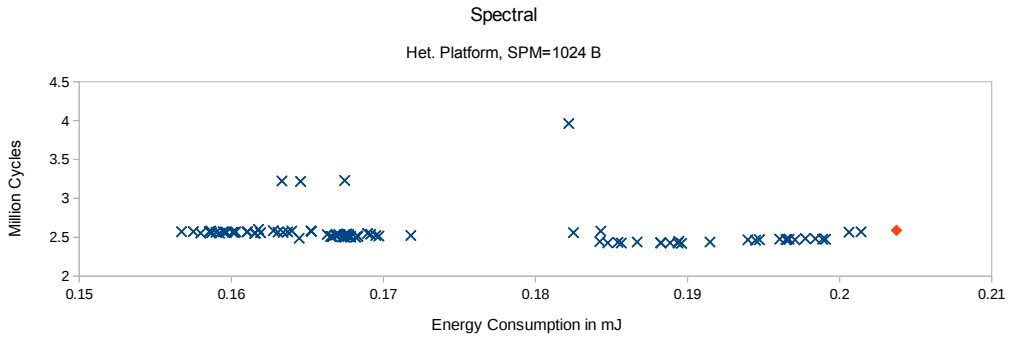


(a) Run time and energy consumption

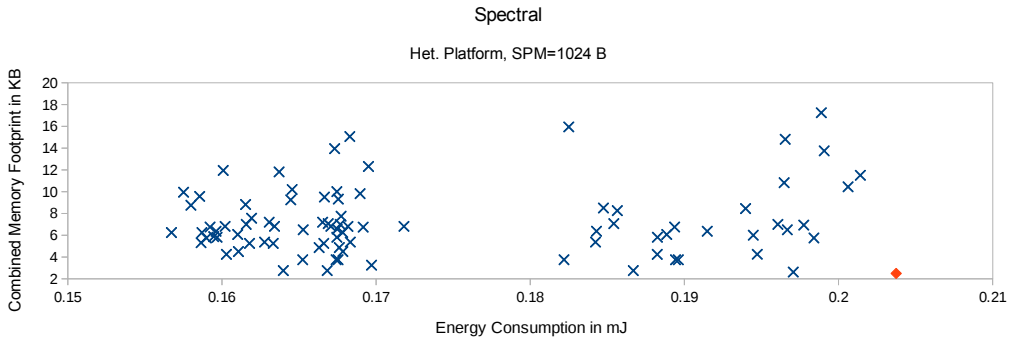


(b) Combined memory footprint and energy consumption

Figure B.10: Spectral on the heterogeneous system with SPM restricted to 512 B.

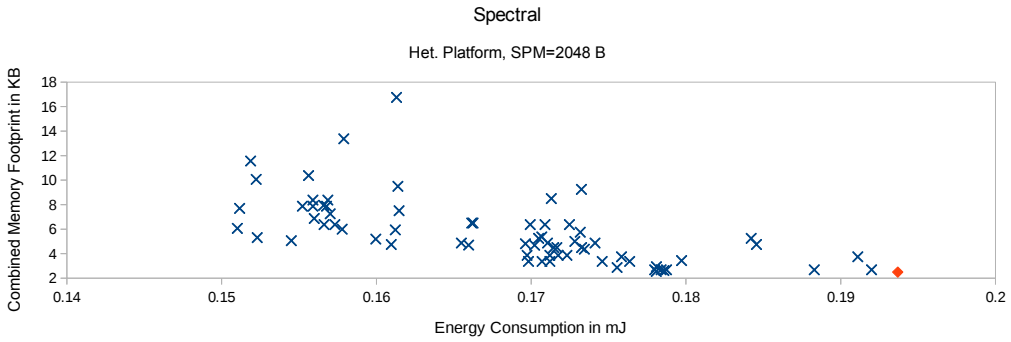


(a) Run time and energy consumption

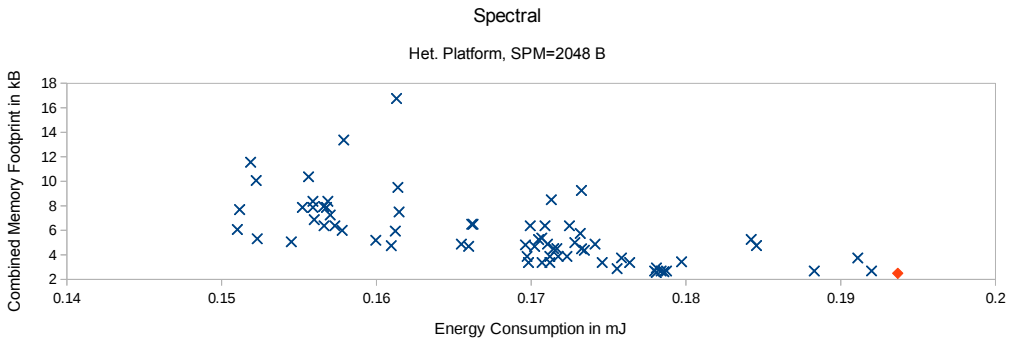


(b) Combined memory footprint and energy consumption

Figure B.11: Spectral on the heterogeneous system with SPM restricted to 1024 B.

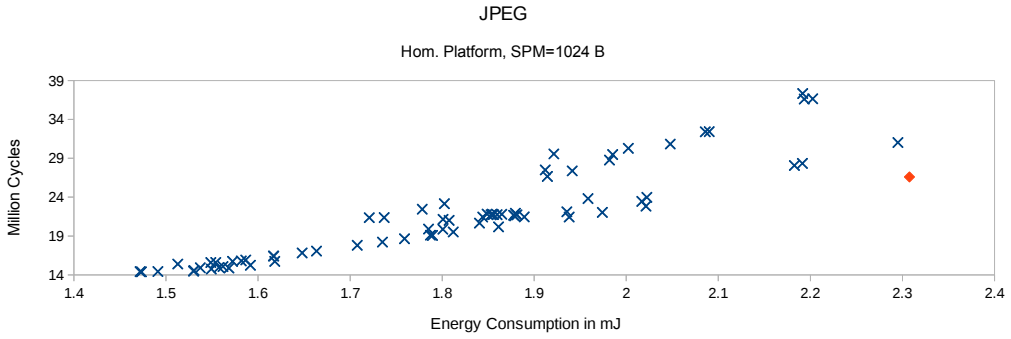


(a) Run time and energy consumption

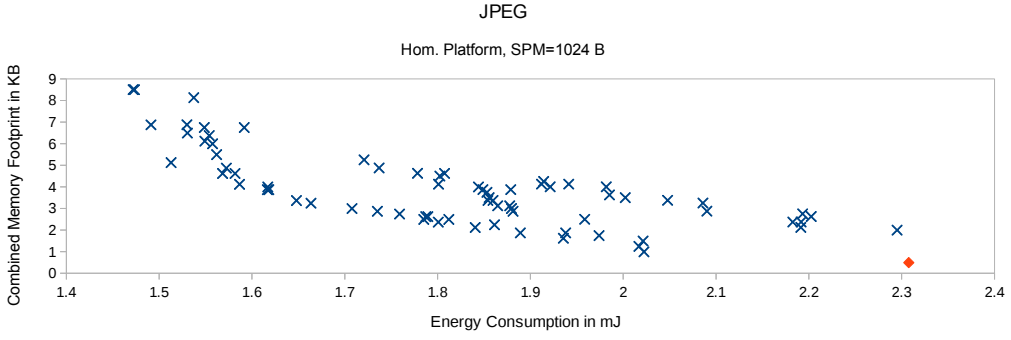


(b) Combined memory footprint and energy consumption

Figure B.12: Spectral on the heterogeneous system with SPM restricted to 2048 B.

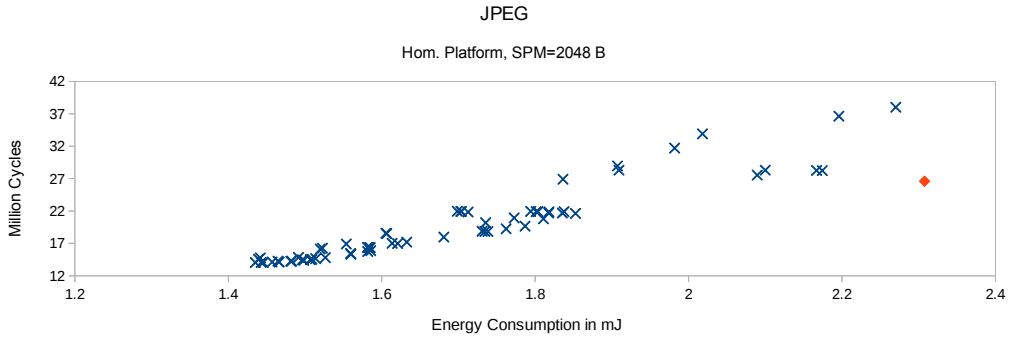


(a) Run time and energy consumption

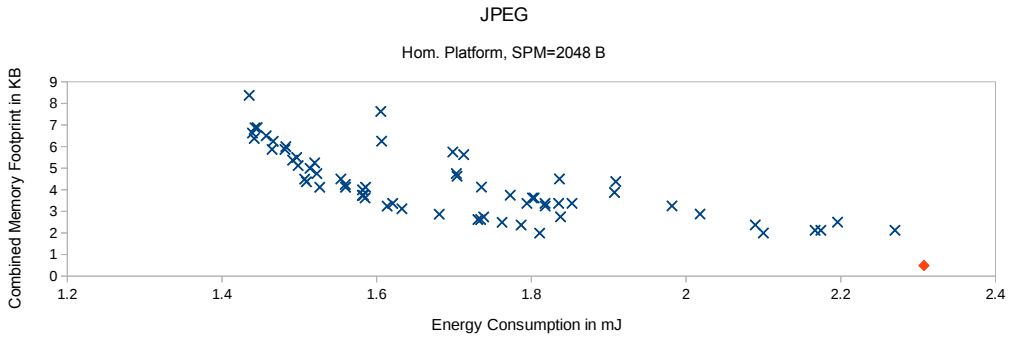


(b) Combined memory footprint and energy consumption

Figure B.13: JPEG on the homogeneous system with SPM restricted to 1024 B.

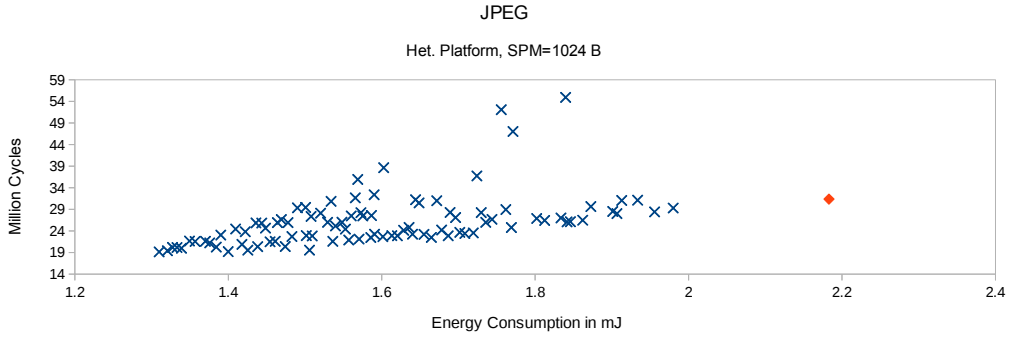


(a) Run time and energy consumption

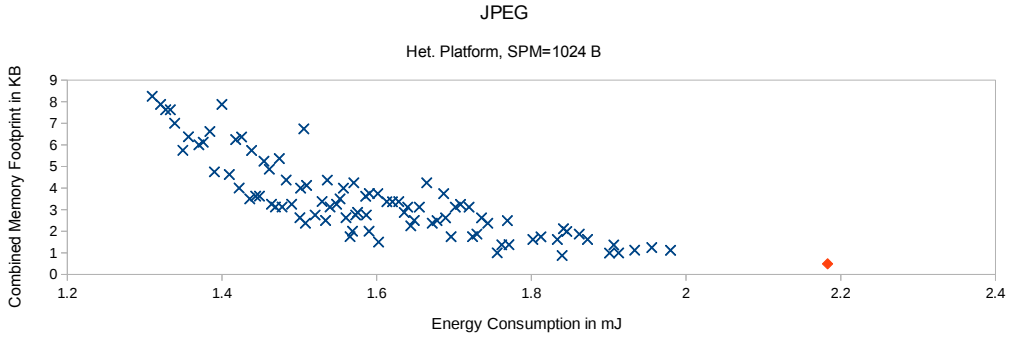


(b) Combined memory footprint and energy consumption

Figure B.14: JPEG on the homogeneous system with SPM restricted to 2048 B.

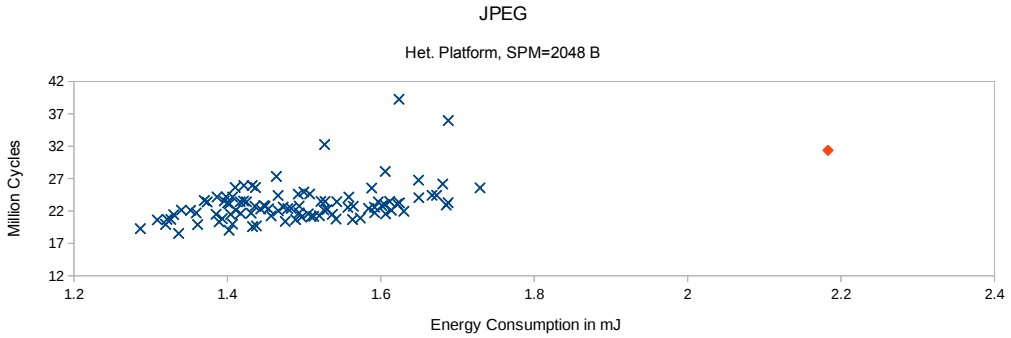


(a) Run time and energy consumption

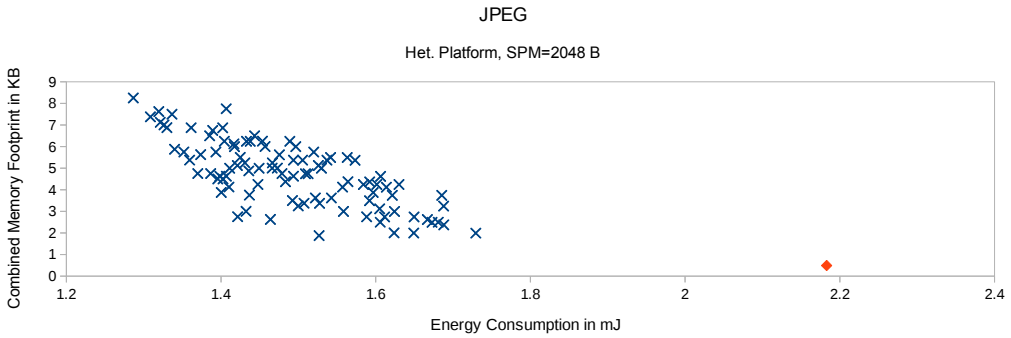


(b) Combined memory footprint and energy consumption

Figure B.15: JPEG on the heterogeneous system with SPM restricted to 1024 B.

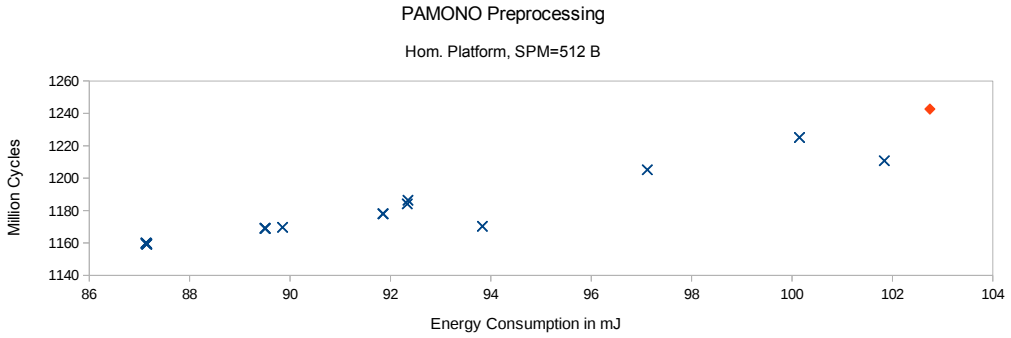


(a) Run time and energy consumption

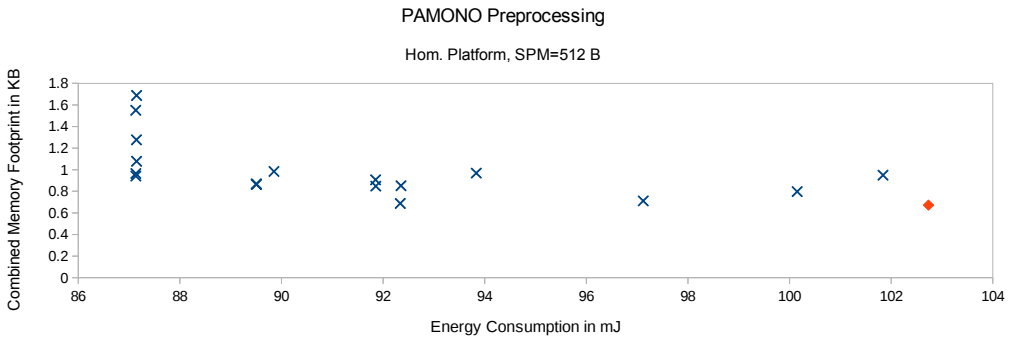


(b) Combined memory footprint and energy consumption

Figure B.16: JPEG on the heterogeneous system with SPM restricted to 2048 B.

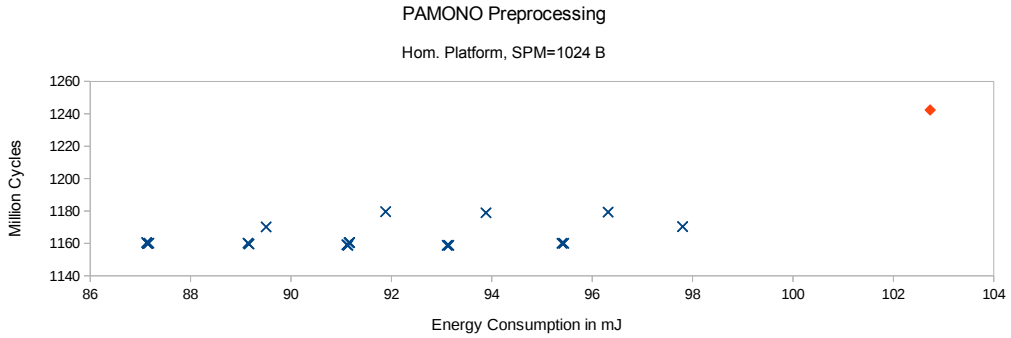


(a) Run time and energy consumption

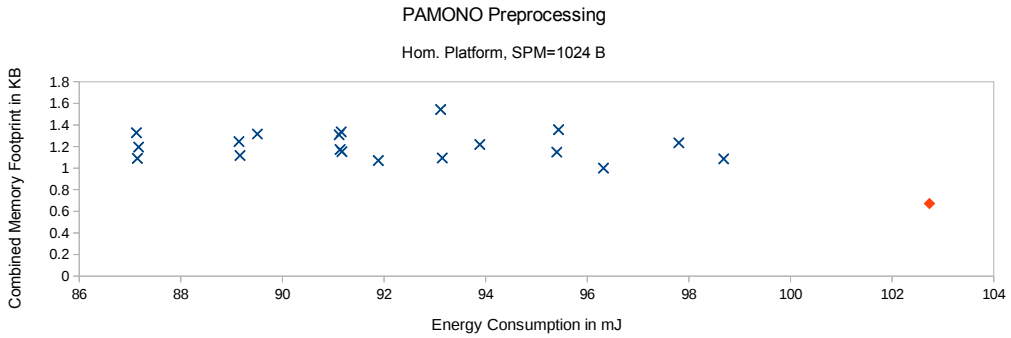


(b) Combined memory footprint and energy consumption

Figure B.17: PAMONO preprocessing on hom. system with SPM restricted to 512 B.



(a) Run time and energy consumption



(b) Combined memory footprint and energy consumption

Figure B.18: PAMONO preprocessing on hom. system with SPM restricted to 1024 B.

Bibliography

[Abb73] Ernst Abbe. "Beiträge zur Theorie des Mikroskops und der mikroskopischen Wahrnehmung". In: *Archiv für mikroskopische Anatomie* 9.1 (12/1873), pp. 413–418. ISSN: 0176-7364. DOI: 10.1007/BF02956173 (Cited on page 113).

[ACP06] Krste Asanovic, Bryan Christopher Catanzaro, David a Patterson, and Katherine a Yelick. *The Landscape of Parallel Computing Research : A View from Berkeley*. Tech. rep. University of California Berkeley, 2006, p. 19. DOI: 10.1145/1562764.1562783 (Cited on page 19).

[Air18] Airbus. *Skywise - Predictive Maintenance*. https://www.airbus.com. 04/2018 (Cited on page 2).

[AKK15] Ismail Akturk, Karen Khatamifard, and Ulya R Karpuzcu. "On Quantification of Accuracy Loss in Approximate Computing". In: *Proceedings of the Workshop on Duplicating, Deconstructing and Debunking*. 2015 (Cited on page 137).

[ALS06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811 (Cited on page 50).

[App17] Apple. *Core ML Framework*. 2017. URL: https://developer.apple.com/documentation/coreml (Cited on page 2).

[App97] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. New York, NY, USA: Cambridge University Press, 1997. ISBN: 0521583896 (Cited on page 50).

[APS04] Giovanni Agosta, Gianluca Palermo, and Cristina Silvano. "Multi-objective co-exploration of source code transformations and design space architectures for low-power embedded systems". In: *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC)*. 2004. ISBN: 1581138121 (Cited on page 130).

[AR12] Ishfaq Ahmad and Sanjay Ranka. *Handbook of Energy-Aware and Green Computing*. Chapman & Hall/CRC, 2012. ISBN: 1466501162, 9781466501164 (Cited on page 130).

[ARM17] ARM. *Q3 2016 Roadshow Slides*. 2017. URL: https://www.arm.com/-/media/arm-com/company/Investors/Quarterly%20Results%20-%20PDFs/ARM_SB_Q3_2016_Roadshow_Slides_FINAL.pdf?la=en (Cited on page 4).

[ASR16] Ali Aalsaud, Rishad Shafik, Ashur Rafiev, Fie Xia, Sheng Yang, and Alex Yakovlev. "Power–Aware Performance Adaptation of Concurrent Applications in Heterogeneous Many-Core Systems". In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED)*. New York, New York, USA: ACM Press, 2016, pp. 368–373. ISBN: 9781450341851. DOI: 10.1145/2934583.2934612 (Cited on page 16).

[AWW09] Krste Asanovic, John Wawrzynek, Katherine Wessel David and Yelick, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, David Morgan Nelson andPatterson, and Koushik Sen. "A view of the parallel computing landscape". In: *Communications of the ACM* 52.10 (2009), p. 56. ISSN: 00010782. DOI: 10.1145/1562764.1562783 (Cited on page 16).

[BBW09] R. Baert, E. Brockmeyer, S. Wuytack, and T.J. Ashby. "Exploring parallelizations of applications for MPSoC platforms using MPA". In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 2009 (Cited on page 34).

[BC10] Woongki Baek and Trishul M. Chilimbi. "Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2010. ISBN: 9781450300193. DOI: 10.1145/1806596.1806620 (Cited on page 136).

[BEN93] Utpal Banerjee, Rudolf Eigenmann, N. Nicolau, David A. Padua, and A. Alexandru. "Automatic Program Parallelization". In: *Proceedings of the IEEE* 81.2 (1993), pp. 211–243. ISSN: 15582256. DOI: 10.1109/5.214548 (Cited on page 34).

[BKS08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*. 2008, p. 72. ISBN: 9781605582825. DOI: 10.1145/1454115.1454128 (Cited on page 144).

[BLT03] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. "PISA — A Platform and Programming Language Independent Interface for Search Algorithms". In: *Evolutionary multi-criterion Optimization*. Vol. 2632. 2003, pp. 494–508. ISBN: 978-3-540-01869-8. DOI: 10.1007/3-540-36970-8_35 (Cited on pages 82, 96).

[BRF15] Luna Backes, Alejandro Rico, and Bjorn Franke. "Experiences in speeding up computer vision applications on mobile computing platforms". In: *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2015, pp. 1–8. ISBN: 978-1-4673-7311-1. DOI: 10.1109/SAMOS.2015.7363653 (Cited on page 15).

[BTM13] Paolo Burgio, Giuseppe Tagliavini, Andrea Marongiu, and Luca Benini. "Enabling Fine-Grained OpenMP Tasking on Tightly-Coupled Shared Memory Clusters". In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 2013, pp. 1504–1509. ISBN: 9781467350716. DOI: 10.7873/DATE.2013.306 (Cited on page 38).

[BVG10] Twan Basten, Emiel Van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian De Smet, Lou Somers, Egbert Teeselink, Nikola Trčka, Frits Vaandrager, Jacques Verriet, Marc Voorhoeve, and Yang Yang. "Model-driven design-space exploration for embedded systems: The octopus toolset". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Vol. 6415 LNCS. PART 1. 2010, pp. 90–105. ISBN: 3642165575. DOI: 10.1007/978-3-642-16558-0_10 (Cited on page 130).

[CCS08] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. "MAPS: An Integrated Framework for MPSoC Application Parallelization". In: *Proceedings of the 45th annual conference on Design automation (DAC)*. New York, New York, USA: ACM Press, 06/2008, pp. 754–759. ISBN: 9781605581156. DOI: 10.1145/1391469.1391663 (Cited on page 35).

[CD14] Tianfeng Chai and Roland R Draxler. "Root mean square error (RMSE) or mean absolute error (MAE)?–Arguments against avoiding RMSE in the literature". In: *Geoscientific Model Development* 7.3 (2014) (Cited on page 138).

[CDV07] Johan Cockx, Kristof Denolf, Bart Vanhoof, and Richard Stahl. "SPRINT: A tool to generate concurrent transaction-level models from sequential code". In: *Eurasip Journal on Advances in Signal Processing* 2007 (2007). ISSN: 16876172. DOI: 10.1155/2007/75373 (Cited on page 34).

[CEM12] Daniel Cordes, Michael Engel, Peter Marwedel, and Olaf Neugebauer. "Automatic extraction of multi-objective aware pipeline parallelism using genetic algorithms". In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*. 10/2012 (Cited on pages 8, 23).

[CEN13a] Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. "Automatic Extraction of Multi-Objective Aware Parallelism for Heterogeneous MPSoCs". In: *Proceedings of the Sixth International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*. 09/2013 (Cited on pages 8, 23).

[CEN13b] Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. "Automatic Extraction of Pipeline Parallelism for Embedded Heterogeneous Multi-Core Platforms". In: *Proceedings of the Sixteenth International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*. 10/2013 (Cited on pages 8, 23).

[CEN13c] Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. "Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs". In: *Proceedings of the Fourth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*. 10/2013 (Cited on pages 8, 23).

[CGF11] Emanuele Cannella, Lorenzo Di Gregorio, Leandro Fiorin, Menno Lindwer, Paolo Meloni, Olaf Neugebauer, and Andy D. Pimentel. "Towards an ESL Design Framework for Adaptive and Fault-tolerant MPSoCs: MADNESS or not?" In: *Proceedings of the 9th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. 10/2011 (Cited on pages 6, 9, 130).

[CGG12] J.M. Cebrian, G.D. Guerrero, and J.M. Garcia. "Energy Efficiency Analysis of GPUs". In: *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. 2012. DOI: 10.1109/IPDPSW.2012.124 (Cited on page 129).

[CHB07] Eric Cheung, Harry Hsieh, and Felice Balarin. "Automatic buffer sizing for rate-constrained KPN applications on multiprocessor system-on-chip". English. In: *Proceedings of the IEEE International High-Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2007, pp. 37–44. ISBN: 1424414806. DOI: 10.1109/HLDVT.2007.4392782 (Cited on page 81).

[CHB09] B. Chapman, Lei Huang, E. Biscondi, E. Stotzer, et al. "Implementing OpenMP on a high performance embedded multicore MPSoC". In: *Proceedings of International Symposium on Parallel and Distributed Processing (IPDPS)*. 2009 (Cited on page 37).

[CL14] Jerónimo Castrillón Mazo and Rainer Leupers. *Programming heterogeneous MPSoCs: Tool flows to close the software productivity gap*. Springer, 2014, pp. 1–232. ISBN: 9783319006758. DOI: 10.1007/978-3-319-00675-8 (Cited on page 35).

[CLA13] Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs". In: *IEEE Transactions on Industrial Informatics* 9.1 (2013), pp. 527–545. ISSN: 15513203. DOI: 10.1109/TII.2011.2173941 (Cited on page 79).

[Con63] Melvin E. Conway. "A multiprocessor system design". In: *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference (AFIPS)*. 1963, p. 139. DOI: 10.1145/1463822.1463838 (Cited on page 29).

[Cor13] Daniel Alexander Cordes. "Automatic Parallelization for Embedded Multi-Core Systems using High-Level Cost Models". PhD thesis. TU Dortmund, 2013 (Cited on pages 8, 13, 19 sq., 23 sq., 66, 163).

[CTL12] Jeronimo Castrillon, Andreas Tretter, Rainer Leupers, and Gerd Ascheid. "Communication-aware mapping of KPN applications onto heterogeneous MPSoCs". In: *Proceedings of the 49th Annual Design Automation Conference on (DAC)*. 06/2012, pp. 1266–1271. ISBN: 9781450311991. DOI: 10.1145/2228360.2228597 (Cited on page 79).

[DAP00] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. "A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II". In: *Parallel Problem Solving from Nature PPSN VI*. Berlin, 2000 (Cited on page 116).

[DGR74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (10/1974), pp. 256–268. ISSN: 0018-9200. DOI: 10.1109/JSSC.1974.1050511 (Cited on page 11).

[DSV06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. "Protothreads". In: *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys)*. June. 2006, p. 29. ISBN: 1595933433. DOI: 10.1145/1182807.1182811 (Cited on page 79).

[EAP06] Karen Egiazarian, Jaakko Astola, Nikolay Ponomarenko, Vladimir Lukin, Federica Battisti, and Marco Carli. “Two New Full-Reference Quality Metrics based on HVS”. In: *Proceedings of the Second International Workshop on Video Processing and Quality Metrics*. 2006, pp. 2–5 (Cited on page 140).

[EBA11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. “Dark silicon and the end of multicore scaling”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. 06/2011, pp. 365–376 (Cited on page 134).

[EEP03] C. Erbas, S.C. Erbas, and A.D. Pimentel. “A multiobjective optimization model for exploring multiprocessor mappings of process networks”. In: *Proceedings of the First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis*. 2003. ISBN: 1-58113-742-7. DOI: 10.1109/CODESS.2003.1275280 (Cited on page 80).

[Eva11] Dave Evans. *The Internet of Things - How the Next Evolution of the Internet Is Changing Everything*. Cisco Internet Business Solutions Group (IBSG), 2011 (Cited on pages 1 sq.).

[FEH18] FEHLER. *Software-based Fault Tolerance for Embedded Real-time Systems*. http://ls12-www.cs.tu-dortmund.de/daes/en/forschung/dependable-embedded-real-time-systems.html. 05/2018 (Cited on pages 13, 53).

[FLP10] Fabrizio Ferrandi, Pier Luca Lanzi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. “Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems”. English. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. Vol. 29. 6. 06/2010, pp. 911–924. DOI: 10.1109/TCAD.2010.2048354 (Cited on page 80).

[GAB98] G. Gogniat, M. Auguin, L. Bianco, and A. Pegatoquet. “Communication synthesis and HW/SW integration for embedded system design”. In: *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES/CASHE)*. 1998, pp. 49–53. ISBN: 0-8186-8442-9. DOI: 10.1109/HSC.1998.666237 (Cited on page 81).

[GAR15] Rem Gensh, Ali Aalsaud, Ashur Rafiev, Fei Xia, Alexei Iliasov, Alexander Romanovsky, and Alex Xakovlev. *Experiments with Odroid-XU3 Board*. Tech. rep. No. CS-TR-1471. Newcastle, 2015 (Cited on page 15).

[GB03] Marc Geilen and Twan Basten. “Requirements on the execution of Kahn process networks”. In: *Programming languages and systems* (2003), pp. 319–334. ISSN: 03029743. DOI: 10.1007/3-540-36575-3_22 (Cited on page 81).

[GMP11] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. “IMPACT: IMPrecise adders for low-power approximate computing”. In: *Proceedings of the International Symposium on Low Power Electronics and Design*. 2011, pp. 409–414. ISBN: 9781612846590. DOI: 10.1109/ISLPED.2011.5993675 (Cited on page 136).

[Gra17] GraphML. http://graphml.graphdrawing.org/. 2017 (Cited on page 50).

[GSt18] GStreamer - open source multimedia framework. https://gstreamer.freedesktop.org/. 04/2018 (Cited on page 118).

[HAA96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih Wei Liao, Edouard Bugnion, and Monica S. Lam. "Maximizing multiprocessor performance with the SUIF compiler". In: *Computer* 29.12 (1996), pp. 84–89. ISSN: 00189162. DOI: 10.1109/2.546613 (Cited on page 34).

[Har16] Hardkernel. *Odroid-XU3*. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127. 06/2016 (Cited on page 14).

[Hei10] Andreas Heinig. *R2G: Supporting POSIX like semantics in a distributed RTEMS system*. Tech. rep. 836. TU Dortmund, Faculty of Computer Science 12, 12/2010 (Cited on pages 8, 14, 166).

[Hei15] Andreas Heinig. "Flexible Error Handling for Embedded Real-Time Systems - Operating System and Run-Time Aspects -". PhD thesis. TU Dortmund, 2015 (Cited on page 13).

[HGT07] Kai Huang Kai Huang, David Grunert, and Lothar Thiele. "Windowed FIFOs for FPGA-based Multiprocessor Systems". In: *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2007, pp. 36–41. ISBN: 978-1-4244-1027-9. DOI: 10.1109/ASAP.2007.4429955 (Cited on page 79).

[HO13] Jie Han and Michael Orshansky. "Approximate Computing: An Emerging Paradigm For Energy-Efficient Design". In: *Proceedings of the European Test Symposium*. 05/2013, pp. 1–6. ISBN: 978-1-4673-6377-8. DOI: 10.1109/ETS.2013.6569370 (Cited on page 136).

[Hol17] Olivera Holzkamp. "Memory-Aware Mapping Strategies for Heterogeneous MPSoC Systems". PhD thesis. TU Dortmund, 2017 (Cited on pages 13, 163).

[HSC11] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. "Dynamic knobs for responsive power-aware computing". In: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*. Vol. 47. 4. New York, New York, USA, 06/2011, pp. 199–212. ISBN: 9781450302661. DOI: 10.1145/1950365.1950390 (Cited on page 136).

[HSH09] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. "Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOs". English. In: *Proceedings of the Workshop on Embedded Systems for Real-Time Multimedia*. 10/2009, pp. 35–44. ISBN: 978-1-4244-5169-2. DOI: 10.1109/ESTMED.2009.5336828 (Cited on pages 79, 109).

[HZ10] Alain Horé and Djemel Ziou. "Image quality metrics: PSNR vs. SSIM". In: *Proceedings of the International Conference on Pattern Recognition*. 2010, pp. 2366–2369. ISBN: 9780769541099. DOI: 10.1109/ICPR.2010.579 (Cited on page 140).

[IAF12] Francisco D Igual, Murtaza Ali, Arnon Friedmann, Eric Stotzer, and Timothy Wentz. "Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 2012 (Cited on page 37).

[IDC15] IDC. *Worldwide Internet of Things Forecast, 2015-2020*. IDC#256397. 2015 (Cited on pages 1 sq.).

[ILK08] Tsuyoshi Isshiki, Dongju Li, and Hiroaki Kunieda. "Multiprocessor SoC Design Framework on Tightly-Coupled Thread Model". In: *Proceedings of the International SoC Design Conference (ISOCC)*. Vol. 1. 2008. ISBN: 9781424425990. DOI: 10.1109/SOCDC.2008.4815572 (Cited on page 35).

[Ima18] Image Processing Place. http://www.imageprocessingplace.com. 04/2018 (Cited on page 147).

[IMM10] Yiannis Iosifidis, Arindam Mallik, Stylianos Mamagkakis, Eddy De Greef, Alexandros Bartzas, Dimitrios Soudris, and Francky Catthoor. "A framework for automatic parallelization, static and dynamic memory optimization in MPSoC platforms". In: *Proceedings of the 47th Design Automation Conference (DAC)*. 2010, p. 549. ISBN: 9781450300025. DOI: 10.1145/1837274.1837410 (Cited on page 34).

[Ind18] Independent JPEG Group. http://www.ijp.org. 04/2018 (Cited on page 147).

[Inf18] Informatik Centrum Dortmund e.V. *ICD-C Compiler framework*. https://icd.de/en/embedded-systems/compiler-tool-development/icd-c. 2018 (Cited on page 40).

[Jen17] Jenkins. https://jenkins.io/. 2017 (Cited on page 62).

[JH07] Woo-Chul Jeun and Soonhoi Ha. "Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS". In: *Proceeding of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2007 (Cited on page 37).

[JM04] Jingcao Hu and R. Marculescu. "Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2004, pp. 234–239. ISBN: 0-7695-2085-5. DOI: 10.1109/DATE.2004.1268854 (Cited on page 81).

[KA02] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-286-0 (Cited on pages 34, 50).

[Kel15] Timon Kelter. "WCET Analysis and Optimization for Multi-Core Real-Time Systems". PhD thesis. TU Dortmund, 2015 (Cited on page 13).

[KKJ08] Seongnam Kwon, Yongjoo Kim, Woo-Chul Jeun, Soonhoi Ha, and Yunheung Paek. "A retargetable parallel-programming framework for MPSoC". In: *ACM Transactions on Design Automation of Electronic Systems* 13.3 (07/2008), pp. 1–18. ISSN: 10844309. DOI: 10.1145/1367045.1367048 (Cited on page 36).

[KKP11] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. "Beyond loop bounds: comparing annotation languages for worst-case execution time analysis". In: *Software & Systems Modeling* 10.3 (07/2011), pp. 411–437. ISSN: 1619-1374. DOI: 10.1007/s10270-010-0161-0 (Cited on page 33).

[KLN17] Helena Kotthaus, Andreas Lang, Olaf Neugebauer, and Peter Marwedel. "R goes Mobile: Efficient Scheduling for Parallel R Programs on Heterogeneous Embedded Systems". In: *Abstract Booklet of the International R User Conference (UseR!)* 07/2017 (Cited on pages 9, 109, 160).

[KMM10] Zvi Kedem, Vincent J Mooney, Kirthi Krishna Muntimadugu, Krishna V. Palem, Avani Devarasetty, and Phani Deepak Parasuramuni. "Optimizing energy to minimize errors in dataflow graphs using approximate adders". In: *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems (CASES)*. 2010, p. 177. ISBN: 9781605589039. DOI: 10.1145/1878921.1878948 (Cited on page 136).

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd. Springer Publishing Company, Incorporated, 2011. ISBN: 1441982361 (Cited on page 2).

[KRB14] Michael Keller, Marius Rosenberg, Malte Brettel, and Niklas Friederichsen. "How Virtualization, Decentrazliation and Network Building Change the Manufacturing Landscape: An Industry 4.0 Perspective". In: 8.1 (2014), pp. 37–44. ISSN: 22128271. DOI: 10.1016/j.procir.2015.02.213. eprint: arXiv:1011.1669v3 (Cited on page 2).

[KT11] Joachim Keinert and Jürgen Teich. *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*. Embedded Systems. Springer, 2011. ISBN: 978-1-4419-7181-4. DOI: 10.1007/978-1-4419-7182-1 (Cited on page 130).

[Kug15] Logan Kugler. "Is "Good Enough" Computing Good Enough?" In: *Communications of the ACM* 58.5 (04/2015), pp. 12–14. ISSN: 00010782. DOI: 10.1145/2742482 (Cited on page 136).

[Küh16] Roland Kühn. "Analysis and evaluation of quality metrics for approximate source-to-source transformations (in German)". Bachelor Thesis. Department of Computer Science, TU Dortmund, 03/2016 (Cited on pages 9, 135).

[KWB10] D. I. Ko, N. Won, and S. S. Bhattacharyya. "Buffer management for multi-application image processing on multi-core platforms: Analysis and case study". In: *Proceeding of the IEEE International Conference on Acoustics, Speech and Signal Processing*. 03/2010, pp. 1662–1665. DOI: 10.1109/ICASSP.2010.5495515 (Cited on page 80).

[LC03] Feng Liu and Vipin Chaudhary. "A practical OpenMP compiler for system on chips". In: *Proceedings of International Workshop on OpenMP Applications and Tools (WOMPAT)*. 2003. ISBN: 3-540-40435-X (Cited on page 37).

[LC08] Ewing Lusk and Anthony Chan. "Early Experiments with the OpenMP/MPI Hybrid Programming Model". In: *Proceedings of 4th International Workshop on OpenMP in a New Era of Parallelism (IWOMP)*. 2008, pp. 36–47. ISBN: 978-3-540-79561-2. DOI: 10.1007/978-3-540-79561-2_4 (Cited on page 38).

[LC12] Jinho Lee and Kiyoung Choi. "Memory-aware mapping and scheduling of tasks and communications on many-core SoC". In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2012, pp. 419–424. ISBN: 9781467307727. DOI: 10.1109/ASPDAC.2012.6164985 (Cited on page 81).

[LDB99] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Anwar Ghuloum, and Monica S. Lam. "SUIF Explorer: An Interactive and Interprocedural Parallelizer". In: *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*. Vol. 34. 8. 1999, pp. 37–48. ISBN: 1581131003. DOI: 10.1145/301104.301108 (Cited on page 34).

[Lee07] Edward A. Lee. "Computing Foundations and Practice for Cyber- Physical Systems : A Preliminary Report". In: (2007), pp. 1–27 (Cited on page 1).

[Lee08] Edward A. Lee. "Cyber Physical Systems: Design Challenges". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 05/2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25 (Cited on pages 2, 62, 94 sq.).

[Lee17] C. G. Lee. *UTDSP Benchmark Suite*. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html. 2017 (Cited on page 45).

[Lei18] Leibniz-Institut für Analytische Wissenschaften - ISAS - e.V. https://www.isas.de/. 04/2018 (Cited on page 114).

[LH07] Chunhua Liao and Oscar Hernandez. "OpenUH: An optimizing, portable OpenMP compiler". In: *Concurrency and Computation Practice and Experience* April (2007), pp. 2317–2332. DOI: 10.1002/cpe (Cited on page 37).

[Lib17] Pascal Libuschewski. "Exploration of Cyber-Physical Systems for GPGPU Computer Vision-Based Detection of Biological Viruses". PhD thesis. 2017 (Cited on pages 113, 116 sq., 119, 123).

[LJV12] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. "RAIDR: Retention-Aware Intelligent DRAM Refresh". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*. Vol. 40. 3. 09/2012, p. 1. ISBN: 978-1-4503-1642-2. DOI: 10.1145/2366231.2337161 (Cited on page 136).

[LMS14] Pascal Libuschewski, Peter Marwedel, Dominic Siedhoff, and Heinrich Müller. "Multi-Objective Energy-Aware GPGPU Design Space Exploration for Medical or Industrial Applications". In: *Proceedings of the Tenth International Conference on Signal-Image Technology and Internet-Based Systems*. 2014 (Cited on page 112).

[LPB18] Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and A Chircop. *ECJ: A java-based evolutionary computation research system*. https://cs.gmu.edu/~eclab/projects/ecj/. 2018 (Cited on page 116).

[LPM11] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G Zorn. "Flikker: saving DRAM refresh-power through critical data partitioning". In: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*. October. 2011, p. 213. ISBN: 9781450302661. DOI: 10.1145/1950365.1950391 (Cited on page 136).

[LSS17] Jan Eric Lenssen, Victoria Shpacovitch, Dominic Siedhoff, Pascal Libuschewski, Roland Hergenröder, and Frank Weichert. "A Review of Nano-Particle Analysis with the PAMONO-Sensor". In: Biosensors: Advances and Reviews, IFSA Publishing, 2017, pp. 81–100 (Cited on pages 112, 114).

[Mar06] Grant Martin. "Overview of the MPSoC design challenge". In: *Proceedings of the 43rd ACM/IEEE Design Automation Conference (DAC)*. 2006, pp. 274–279. ISBN: 1408327732 (Cited on page 28).

[Mar17] Peter Marwedel. *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems and the Internet of Things*. 3rd. ISBN 978-94-007-0256-1. Springer, 2017 (Cited on pages 1–4, 9).

[MB09] A. Marongiu and L. Benini. "Efficient OpenMP support and extensions for MPSoCs with explicitly managed memory hierarchy". In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 2009 (Cited on page 38).

[MB12] A. Marongiu and L. Benini. "An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs". In: *IEEE Transactions on Computers* 61.2 (2012). ISSN: 0018-9340 (Cited on page 38).

[MBA09] Jean Yves Mignolet, Rogier Baert, Thomas J. Ashby, Prabhat Avasare, Hye On Jang, and Jae Cheol Son. "MPA: Parallelizing an application onto a multicore platform made easy". In: *IEEE Micro* 29.3 (2009), pp. 31–39. ISSN: 02721732. DOI: 10.1109/MM.2009.46 (Cited on page 34).

[MBJ09] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. *CACTI 6.0: A Tool to Model Large Caches*. Tech. rep. HP Laboratories, 2009. URL: http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html (Cited on pages 13, 163).

[MCB15] James Manyika, Michael Chui, Peter Bisson, Jonathan Woetzel, Richard Dobbs, Jacques Bughin, and Dan Aharon. "The Internet of Things: Mapping the value beyond the hype". In: June (2015), p. 144. ISSN: 1860949X. DOI: 10.1007/978-3-319-05029-4_7 (Cited on page 2).

[MCT15] Andrea Marongiu, Alessandro Capotondi, Giuseppe Tagliavini, and Luca Benini. "Simplifying Many-Core-Based Heterogeneous SoC Programming With Offload Directives". In: *IEEE Transactions on Industrial Informatics* 11.4 (2015) (Cited on page 38).

[MEM04] MEMSIC Inc. *TelosB datasheet: 6020-0094-03 rev*. Tech. rep. 2004, pp. 1–2 (Cited on page 147).

[MFN17] Peter Marwedel, Heiko Falk, and Olaf Neugebauer. "Memory-Aware Optimization of Embedded Software for Multiple Objectives". In: *Handbook of Hardware/Software Codesign*. Ed. by Soonhoi Ha and Jürgen Teich. Springer Netherlands, 2017. ISBN: 978-94-017-7358-4. DOI: 10.1007/978-94-017-7358-4_27-2 (Cited on pages 9, 113).

[Mid12] Samuel P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Vol. 7. 2012, pp. 1–169. ISBN: 9781608458417. DOI: 10.2200/S00340ED1V01Y201201CAC019 (Cited on page 34).

[Mit16] Sparsh Mittal. "A Survey of Techniques for Approximate Computing". In: *ACM Computing Surveys* 48.4 (03/2016), pp. 1–33. ISSN: 03600300. DOI: 10.1145/2893356. arXiv: arXiv:1011.1669v3 (Cited on page 136).

[MMB11] Arindam Mallik, Stylianos Mamagkakis, Christos Baloukas, Lazaros Papadopoulos, Dimitrios Soudris, Sander Stuijk, Olivera Jovanovic, Florian Schmoll, Daniel Cordes, Robert Pyka, Peter Marwedel, et al. "MNEMEE – An automated toolflow for parallelization and memory management in MPSoC platforms". In: (06/2011) (Cited on pages 6, 13).

[MOK10] Masayoshi Mase, Yuto Onozaki, Keiji Kimura, and Hironori Kasahara. "Parallelizable C and Its Performance on Low Power High Performance Multicore Processors". In: *Proceedings of the 15th Workshop on Compilers for Parallel Computing*. 2010 (Cited on page 33).

[MPI17] MPI Forum. *Standardization Forum for the Message Passing Interface (MPI)*. `http://mpi-forum.org/`. 2017 (Cited on page 38).

[MPN02] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation". In: *Proceedings of the joint conference on Languages, compilers and tools for embedded systems software and compilers for embedded systems (LCTES/SCOPES)*. 2002, p. 18. ISBN: 1581135270. DOI: `10.1145/513829.513835` (Cited on page 130).

[MR13] Katharina Morik and Wolfgang Rhode (Editors). *Technical report for Collaborative Research Center SFB 876 - Graduate School*. Tech. rep. 4. TU Dortmund University, 10/2013 (Cited on page 9).

[MR14] Katharina Morik and Wolfgang Rhode (Editors). *Technical report for Collaborative Research Center SFB 876 - Graduate School*. Tech. rep. 10. TU Dortmund University, 12/2014 (Cited on page 9).

[MR15] Katharina Morik and Wolfgang Rhode (Editors). *Technical report for Collaborative Research Center SFB 876 - Graduate School*. Tech. rep. 3. TU Dortmund University, 12/2015 (Cited on page 9).

[MR16] Katharina Morik and Wolfgang Rhode (Editors). *Technical report for Collaborative Research Center SFB 876 - Graduate School*. Tech. rep. 7. TU Dortmund University, 12/2016 (Cited on page 9).

[MR17] Katharina Morik and Wolfgang Rhode (Editors). *Technical report for Collaborative Research Center SFB 876 - Graduate School*. Tech. rep. 2. TU Dortmund University, 12/2017 (Cited on page 9).

[MSH10] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. "Quality of service profiling". In: *Proceedings of the International Conference on Software Engineering*. Vol. 1. 2010, pp. 25–34. ISBN: 978-1-60558-719-6. DOI: `10.1145/1806799.1806808` (Cited on pages 136, 144).

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4 (Cited on page 50).

[Mul17] Multicore Association. `https://www.multicore-association.org/`. 2017 (Cited on page 37).

[MV14] Sparsh Mittal and Jeffrey S Vetter. "A Survey of Methods For Analyzing and Improving GPU Energy Efficiency". In: *ACM Comput. Surv.* 47.2 (2014), pp. 1–22. ISSN: 03600300. DOI: `10.1145/2636342`. arXiv: `1404.4629` (Cited on page 129).

[NBZ15] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H.J. Kelly, Andrew J. Davison, Mikel Luján, Michael F.P. O'Boyle, Graham Riley, Nigel Topham, and Steve Furber. "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM". In: *Proceedings of the IEEE International Conference on Robotics and Automation*. Vol. 2015-June. June. 2015, pp. 5783–5790. ISBN: 9781479969234. DOI: `10.1109/ICRA.2015.7140009`. arXiv: `1410.2167` (Cited on page 15).

[NEM14] Olaf Neugebauer, Michael Engel, and Peter Marwedel. "A Parallelization Approach for Resource-Restricted Embedded Heterogeneous MPSoCs Inspired by OpenMP". In: *The 1st ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems (SEPS)*. 10/2014 (Cited on pages 9, 29).

[NEM15a] Olaf Neugebauer, Michael Engel, and Peter Marwedel. "Approximate Communication in Parallel Applications for Resource-Constrained Embedded Systems". In: *Workshop on Approximate Computing*. 10/2015 (Cited on pages 9, 135, 152).

[NEM15b] Olaf Neugebauer, Michael Engel, and Peter Marwedel. "Multi-Objective Aware Communication Optimization for Resource-Restricted Embedded Systems". In: *Proceedings of Architecture of Computing Systems (ARCS)*. 2015 (Cited on pages 76, 84 sq.).

[NEM16] Olaf Neugebauer, Michael Engel, and Peter Marwedel. "A Parallelization Approach for Resource-Restricted Embedded Heterogeneous MPSoCs Inspired by OpenMP". In: *Journal of Systems and Software (JSS)* 125 (2016), pp. 439–448. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/j.jss.2016.08.069 (Cited on pages 9, 29).

[NLE15] Olaf Neugebauer, Pascal Libuschewski, Michael Engel, Heinrich Müller, and Peter Marwedel. "Plasmon-based Virus Detection on Heterogeneous Embedded Systems". In: *Proceedings of Workshop on Software & Compilers for Embedded Systems (SCOPES)*. 2015, pp. 48–57. ISBN: 978-1-4503-3593-5. DOI: 10.1145/2764967.2764976 (Cited on pages 9, 113–116, 118 sq., 122, 126 sqq., 165).

[NMK17] Olaf Neugebauer, Peter Marwedel, Roland Kühn, and Michael Engel. "Quality Evaluation Strategies for Approximate Computing in Embedded Systems". In: *Technological Innovation for Smart Systems*. Ed. by Luis M. Camarinha-Matos, Mafalda Parreira-Rocha, and Javaneh Ramezani. Vol. 499. Cham: Springer International Publishing, 2017, pp. 203–210. ISBN: 978-3-319-56077-9 (Cited on pages 9, 135, 141, 146).

[NMS09] Dmitry Nadezhkin, Sjoerd Meijer, Todor Stefanov, and Ed Deprettere. "Realizing FIFO communication when mapping kahn process networks onto the cell". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 5657 LNCS. 2009, pp. 308–317. ISBN: 3642031374. DOI: 10.1007/978-3-642-03138-0_34 (Cited on page 79).

[Nol14] Stefan Noll. "Exploration of the usability of embedded multi-core platforms for virus detection software with the PAMONO sensor (in German)". Bachelor Thesis. Department of Computer Science, TU Dortmund, 09/2014 (Cited on pages 9, 62, 65).

[NTS08] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. "Daedalus: Toward composable multimedia MP-SoC design". In: *Proceedings of the 45th Design Automation Conference (DAC)*. 2008, pp. 574–579. ISBN: 9781605581156. DOI: 10.1109/DAC.2008.4555882 (Cited on pages 36, 130).

[OCV13] Maximilian Odendahl, Jeronimo Castrillon, Vitaliy Volevach, Rainer Leupers, and Gerd Ascheid. "Split-cost communication model for improved MPSoC application mapping". In: *Proceedings of the International Symposium on System on Chip (SoC)*. 10/2013, pp. 1–8. ISBN: 978-1-4799-1191-2. DOI: 10.1109/ISSoC.2013.6675280 (Cited on page 80).

[Ope17] OpenMP. *The OpenMP API specification for parallel programming*. http://www.openmp.org/. 2017 (Cited on pages 29, 37).

[ORS05] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. "Automatic Thread Extraction with Decoupled Software Pipelining". In: *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*. 2005, pp. 105–116. ISBN: 0769524400. DOI: 10.1109/MICRO.2005.13 (Cited on page 34).

[Par95] Thomas M. Parks. "Bounded Scheduling of Process Networks". PhD thesis. University of California at Berkeley, 1995. URL: http://ptolemy.eecs.berkeley.edu/publications/papers/95/parksThesis/ (Cited on page 80).

[PC11] Antoniu Pop and Albert Cohen. "A stream-computing extension to OpenMP". In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*. 2011, p. 5. ISBN: 9781450302418. DOI: 10.1145/1944862.1944867 (Cited on page 38).

[PC13] Antoniu Pop and Albert Cohen. "OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (01/2013). ISSN: 1544-3566 (Cited on page 38).

[Pet13] Peter Greenhalgh, ARM. *Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf. 2013 (Cited on page 14).

[PG02] M. Palesi and T. Givargis. "Multi-objective Design Space Exploration using Genetic Algorithms". In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. (CODES)*. 2002, pp. 67–72. ISBN: 1-58113-542-4. DOI: 10.1109/CODES.2002.1003603 (Cited on page 130).

[Pim17] Andy D Pimentel. "Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration". In: *IEEE Design & Test* 34.1 (02/2017), pp. 77–90. ISSN: 2168-2356. DOI: 10.1109/MDAT.2016.2626445 (Cited on page 130).

[PKS13] Hana Park, Young Woong Ko, Jungmin So, and Jeong-Gun Lee. "Performance/Power Design Space Exploration and Analysis for GPU Based Software". In: *International Journal of Control and Automation* 6.6 (2013), pp. 371–380. ISSN: 20054297. DOI: 10.14257/ijca.2013.6.6.35 (Cited on page 131).

[PSK13] Nam Khanh Pham, Amit Kumar Singh, Akash Kumar, and Khin Mi Mi Aung. "Incorporating Energy and Throughput Awareness in Design Space Exploration and Run-Time Mapping for Heterogeneous MPSoCs". In: *Proceedings of the Euromicro Conference on Digital System Design*. 09/2013, pp. 513–521. DOI: 10.1109/DSD.2013.61 (Cited on page 130).

[PWI15] Alok Prakash, Siqi Wang, Alexandru Eugen Irimiea, and Tulika Mitra. "Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms". In: *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD)*. 10/2015, pp. 208–215. ISBN: 978-1-4673-7166-7. DOI: 10.1109/ICCD.2015.7357105 (Cited on page 15).

[Pyk17] Robert Pyka. "Memory-aware platform description and framework for source-level embedded MPSoC software optimization". PhD thesis. 2017 (Cited on pages 8, 13, 21).

[RFG08] Alastair D. Reid, Krisztian Flautner, Edmund Grimley-Evans, and Yuan Lin. "SoC-C: Efficient Programming Abstractions for Heterogeneous Multicore Systems on Chip". In: *Compilers, Architectures and Synthesis for Embedded Systems*. 2008, pp. 95–104. ISBN: 9781605584690. DOI: 10.1145/1450095.1450112 (Cited on page 36).

[RHJ09] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes". In: *Proceedings of 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43 (Cited on page 38).

[Rin06] Martin Rinard. "Probabilistic Accuracy Bounds for Fault-Tolerant Computations that Discard Tasks". In: *Proceedings of the International Conference on Supercomputing*. 2006, pp. 324–334. ISBN: 1595932828. DOI: 10.1145/1183401.1183447 (Cited on page 151).

[Rin07] Martin C. Rinard. "Using early phase termination to eliminate load imbalances at barrier synchronization points". In: *Proceedings of the International Conference on Object-Oriented Programming Systems Languages & Applications*. Vol. 42. 10. 2007, p. 369. ISBN: 9781595937865. DOI: 10.1145/1297105.1297055 (Cited on page 151).

[RLS10] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. "Cyber-physical systems: The Next Computing Revolution". In: *Proceedings of the 47th Design Automation Conference (DAC)*. 2010, pp. 731–736. ISBN: 9781450300025. DOI: 10.1145/1837274.1837461 (Cited on page 3).

[RSN12] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. "Programming with relaxed synchronization". In: *Proceedings of the Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*. 2012, p. 41. ISBN: 9781450316323. DOI: 10.1145/2414729.2414737 (Cited on pages 137, 152).

[RTE16] RTEMS. *RTEMS Operating System | Real-Time and Real Free*. http://www.rtems.com/. 2016 (Cited on page 13).

[SCZ15] Peng Sun, Sunita Chandrasekaran, Suyang Zhu, and Barbara Chapman. "Deploying OpenMP task parallelism on multicore embedded systems with MCA task APIs". In: *Proceedings of the 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems, (HPCC-CSS-ICESS)*. 2015, pp. 843–847. ISBN: 9781479989362. DOI: 10.1109/HPCC-CSS-ICESS.2015.88 (Cited on page 37).

[SDF11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "EnerJ". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. 2011, p. 164. ISBN: 9781450306638. DOI: 10.1145/1993498.1993518 (Cited on page 136).

[SFB17a] SFB 876 - B2. *Resource optimizing real time analysis of artifactious image sequences for the detection of nano objects*. 2017. URL: http://sfb876.tu-dortmund.de/SPP/sfb876-b2.html (Cited on pages 4, 6).

[SFB17b] SFB 876 - Software. *Resource optimizing real time analysis of artifactious image sequences for the detection of nano objects*. 2017. URL: http://sfb876.tu-dortmund.de/auto?self=Software (Cited on pages 15, 142, 164).

[Sie16] Dominic Siedhoff. "A Parameter-Optimizing Model-Based Approach to the Analysis of Low-SNR Image Sequences for Biological Virus Detection". PhD thesis. TU Dortmund, 2016 (Cited on pages 116 sq.).

[Sie17] Siemens. *Embedded Multicore Building Blocks (EMB^2)*. `https://github.com/siemens/embb`. 2017 (Cited on page 37).

[SJA13] Eric Stotzer, Ajay Jayaraj, Murtaza Ali, Arnon Friedmann, Gaurav Mitra, Alistair P. Rendell, and Ian Lintault. "OpenMP on the low-power TI keystone II ARM/DSP system-on-chip". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8122 LNCS. 6. 2013, pp. 114–127. ISBN: 9783642406973. DOI: `10.1007/978-3-642-40698-0_9`. arXiv: `arXiv:1302.5679v1` (Cited on page 37).

[SKW01] Stefan Steinke, Markus Knauer, Lars Wehmeyer, and Peter Marwedel. "An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations". In: *Proceedings of the 11th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2001. DOI: `10.1.1.115.3528` (Cited on pages 13, 163).

[SLJ14] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. "Scaling Performance via Self-Tuning Approximation for Graphics Engines". In: *ACM Transactions on Computer Systems (TOCS)* 32.3 (2014) (Cited on page 136).

[SMH11] Stelios Sidiroglou, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. "Managing performance vs. accuracy trade-offs with loop perforation". In: *Proceedings of the Foundations of Software Engineering*. 2011, pp. 124–134. ISBN: 9781450304436. DOI: `10.1145/2025113.2025133` (Cited on pages 136, 144).

[SNU17] SNU Real-time Benchmark Suite. `http://www.cprover.org/goto-cc/examples/snu.html`. 2017 (Cited on pages 62, 94 sq.).

[SSL17] Victoria Shpacovitch, Irina Sidorenko, Jan Eric Lenssen, Vladimir Temchura, Frank Weichert, Heinrich Müller, Klaus Überla, Alexander Zybin, Alexander Schramm, and Roland Hergenröder. "Application of the PAMONO-sensor for quantification of microvesicles and determination of nano-particle size distribution". In: *Sensors (Switzerland)* 17.2 (2017). ISSN: 14248220. DOI: `10.3390/s17020244` (Cited on pages 112, 114).

[SSO14] Weihua Sheng, Stefan Schürmans, Maximilian Odendahl, Mark Bertsch, Vitaliy Volevach, Rainer Leupers, and Gerd Ascheid. "A compiler infrastructure for embedded heterogeneous MPSoCs". In: *Parallel Computing* 40.2 (2014), pp. 51–68. ISSN: 01678191. DOI: `10.1016/j.parco.2013.11.007` (Cited on page 35).

[Syn17] Synopsys. *Virtualizer, Virtual Prototyping Solution*. `http://www.synopsys.com`. 2017 (Cited on page 13).

[SZS14] Dominic Siedhoff, Alexander Zybin, Victoria Shpacovitch, and Pascal Libuschewski. *PAMONO Sensor Data*. 2014. DOI: `10.15467/e9ofylrdvk` (Cited on pages 124, 147).

[TCA07] W. Thies, V. Chandrasekhar, and S. Amarasinghe. "A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs". In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2007 (Cited on page 35).

[Tex13] Texas Instruments Incorporated. *High- or Low-Side Measurement, Bidirectional CURRENT/POWER MONITOR with 1.8-V I^2C Interface*. SBOS644-FEBRUARY 2013. Texas Instruments Incorporated. 02/2013 (Cited on pages 15, 123, 164).

[Tex18] Texas Instruments. *Keystone Device Architecture*. `http://processors.wiki.ti.com/index.php/Keystone_Device_Architecture`. 2018 (Cited on page 12).

[TF10] Georgios Tournavitis and Björn Franke. "Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information". In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*. 2010, pp. 377–388. ISBN: 978-1-4503-0178-7. DOI: `10.1145/1854273.1854321` (Cited on page 34).

[TWF09] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP MFP O'Boyle. "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping". In: *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*. 2009, pp. 177–187. ISBN: 9781605583921. DOI: `10.1145/1543135.1542496` (Cited on page 34).

[VAR11] Rangharajan Venkatesan, Amit Agarwal, Kaushik Roy, and Anand Raghunathan. "MACACO: Modeling and analysis of circuits for approximate computing". In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers (ICCAD)*. 2011, pp. 667–673. ISBN: 9781457713989. DOI: `10.1109/ICCAD.2011.6105401` (Cited on page 136).

[VCR15a] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. "Approximate computing and the quest for computing efficiency". In: *Proceedings of the 52nd Annual Design Automation Conference on (DAC)*. 2015, pp. 1–6. ISBN: 9781450335201. DOI: `10.1145/2744769.2751163` (Cited on page 136).

[VCR15b] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. "Computing approximately, and efficiently". In: *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 748–751. ISBN: 9783981537048 (Cited on page 136).

[Vid16] VideoLAN. *x264,the best H.264/AVC encoder*. `http://www.videolan.org/developers/x264.html`. 06/2016 (Cited on page 144).

[VNS07] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. "pn: A Tool for Improved Derivation of Process Networks". In: *EURASIP Journal on Embedded Systems* 2007.1 (01/2007), pp. 1–13. ISSN: 1687-3955. DOI: `10.1155/2007/75947` (Cited on pages 36, 80).

[WB02] Zhou Wang and Alan C Bovik. "A universal image quality index". In: *Signal Processing Letters, IEEE* 9.3 (2002) (Cited on page 139).

[WB09] Zhou Wang and Alan C Bovik. "Error : Love It or Leave It ?" In: *IEEE Signal Processing Magazine* 26.January (2009), pp. 98–117. ISSN: 1053-5888. DOI: `10.1109/MSP.2008.930649` (Cited on page 138).

[WBS04] Zhou Wang, Alan C Bovik, Hamid Rahim Sheikh, and Eero P. Simoncelli. "Image quality assessment: From error visibility to structural similarity". In: *IEEE Transaction on Image Processing* 13.4 (2004). ISSN: 10577149 (Cited on page 140).

[WC13] Cheng Wang and Sunita Chandrasekaran. "libEOMP: a portable OpenMP runtime library based on MCA APIs for embedded systems". In: *Proceedings of International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. 2013 (Cited on page 37).

[WFW94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers". In: *ACM SIGPLAN Notices* 29.12 (1994), pp. 31–37. ISSN: 03621340. DOI: 10.1145/193209.193217 (Cited on page 34).

[WLB04] Zhou Wang, Ligang Lu, and Alan C Bovik. "Video quality assessment based on structural distortion measurement". In: *Signal processing: Image communication* 19.2 (2004) (Cited on page 143).

[WM05] Cort J Willmott and Kenji Matsuura. "Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance". In: *Climate research* 30.1 (2005) (Cited on page 138).

[Xip18] Xiph.Org Foundation. http://www.xiph.org. 05/2018 (Cited on pages 144, 147).

[XMK16] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. "Approximate Computing: A Survey". In: *IEEE Design & Test* 33.1 (02/2016), pp. 8–22. ISSN: 2168-2356. DOI: 10.1109/MDAT.2015.2505723 (Cited on page 136).

[yWo17] yWorks. *yEd Graph Editor*. https://www.yworks.com/products/yed. 2017 (Cited on page 50).

[Za10] A Zybin and et al. "Real-time Detection of Single Immobilized Nanoparticles by Surface Plasmon Resonance Imaging". In: *Plasmonics* 5 (2010), pp. 31–35 (Cited on pages 112, 114).

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. "SPEA2: Improving the Strength Pareto Evolutionary Algorithm". In: *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems* (2001), pp. 95–100. ISSN: 03772217. DOI: 10.1.1.28.7571. arXiv: arXiv:1011.1669v3 (Cited on page 96).

List of Figures

List of Tables

List of Listings

List of Algorithms

Index