2024 Annual Report of the Chair of Computer Science 2 (Programming Systems)

1 Staff

Julian Brandner, M. Sc., Tobias Heineken, M. Sc., Hon.-Prof. Dr.-Ing. Bernd Hindel, Hon.-Prof. Dr.-Ing. Detlef Kips, Florian Mayer, M. Sc. (until July 31, 2024), Dr.-Ing. Norbert Oster, Akad. ORat, Prof. Dr. Michael Philippsen (Ordinarius), Prof. em. Dr. Hans Jürgen Schneider (Emeritus), David Schwarzbeck, M. Sc. (since October 1, 2024), Alma Sinanović (IT-support, since June 1, 2024), Margit Zenk (Secretary).

Guests and external teaching staff: Jonas Butz, M. Sc. (associate lecturer), Dr.-Ing. Klaudia Dussa-Zieger (associate lecturer), Dr.-Ing. Tobias Feigl (associate lecturer), Dr.-Ing. Martin Jung (associate lecturer), Patrick Kreutzer, M. Sc., Dipl.-Inf. Daniela Novac.

2 Overview

We develop scientific solutions for software engineers in industry who work on **parallel software** for multicores and for distributed or embedded systems made thereof. We take a **code-centric approach**, construct operational **prototypes**, and **evaluate** them both quantitatively and qualitatively. Corner stones of our field of research:
(a) We work on **programming models** for **heterogeneous** parallelism, from which we then generate portable and efficient code for multicores, GPUs, accelerators, mobile devices, FPGAs, etc.
(b) We help parallelize software for multicores. Our tools analyze code repositories and help developers in **migrating** and **refactoring** projects.
(c) We analyze code. Our **code analysis tools** are fast, interactive, incremental and sometimes work in parallel themselves. They not only detect race conditions, conflicting accesses to resources, etc. The resulting suggestions on how to improve the code also show up in the IDE where they matter.
(d) We **test** parallel code and **diagnose** the root causes of problems. Our tools generate test data, track down erratic runtime behavior, and prevent **authenticity attacks**.

3 Research projects

3.1 AutoCompTest – *Automatic Testing of Compilers*

Compilers for programming languages are very complex applications and their correctness is crucial: If a compiler is erroneous (i.e., if its behavior deviates from that defined by the language specification), it may generate wrong code or crash with an error message. Often, such errors are hard to detect or circumvent. Thus, users typically demand a bug-free compiler implementation.

Unfortunately, research studies and online bug databases suggest that probably no real compiler is bug-free. Several research works therefore aim to improve the quality of compilers. Since the formal verification (i.e., a proof of a compiler's correctness) is often prohibited in practice, most of the recent works focus on techniques for extensively testing compilers in an automated way. For this purpose, the compiler under test is usually fed with a test program and its behavior (or that of the generated program) is checked: If the actual behavior does not match the expectation (e.g., if the compiler crashes when fed with a valid test program), a compiler bug has been found. If this testing process is to be carried out in a fully automated way, three main challenges arise:

- Where do the test programs come from that are fed into the compiler?
- What is the expected behavior of the compiler or its output program? How can one determine if the compiler worked correctly?
- How can test programs that indicate an error in the compiler be prepared to be most helpful in fixing the error in the compiler?

While the scientific literature proposes several approaches for dealing with the second challenge (which are also already established in practice), the automatic generation of random test programs still remains a challenge. If all parts of a compiler should be tested, the test programs have to conform to all rules of the respective programming language, i.e., they have to be syntactically and semantically correct (and thus compilable). Due to the large number of rules of "real" programming languages, the generation of such compilable programs is a non-trivial task. This is further complicated by the fact that the program generation has to be as efficient as possible: Research suggests that the efficiency of such an approach significantly impacts its effectivity – in a practical scenario, a tool can only be used for detecting compiler bugs if it can generate many (and large) programs in short time.

The lack of an appropriate test program generator and the high costs associated with the development of such a tool often prevent the automatic testing of compilers in practice. Our research project therefore aims to reduce the effort for users to implement efficient program generators.

Large programs generated by efficient automatic generation of random test programs are difficult to use for debugging. Typically, only a small part of the program is the cause of the error, and as many other parts as possible must be automatically removed before the error can be corrected. This so-called test case reduction also uses the solutions already mentioned for detecting the expected behavior so that a joint consideration makes sense. Test case reduction is an essential component for automatically generated programs and should be designed to process error-triggering programs from all sources.

Unfortunately, it is often unclear which of the various methods presented in the scientific literature is best suited to a particular situation. Additionally, test case reduction can be a time-consuming process. Our research project aims to create a significant collection of unreduced test cases and to use them to compare and improve existing procedures.

In 2018, we started the development of such a tool. As input, it requires a specification of a programming language's syntactic and semantic rules by means of an abstract attribute grammar. Such a grammar allows for a short notation of the rules on a high level of abstraction. Our newly devised algorithm then generates test programs that conform to all of the specified rules. It uses several novel technical ideas to reduce its expected runtime. This way, it can generate large sets of test programs in acceptable time, even when executed on a standard desktop computer. A first evaluation of our approach did not only show that it is efficient and effective, but also that it is versatile. Our approach detected several bugs in the C compilers gcc and clang (and achieved a bug detection rate which is comparable to that of a state-of-the-art C program generator from the literature) as well as multiple bugs in different SMT solvers. Some of the bugs that we detected were previously unknown to the respective developers.

In 2019, we implemented additional features for the definition of language specifications and improved the efficiency of our program generator. These two contributions considerably increased the throughput of our tool. By developing additional language specifications, we were also able to uncover bugs in compilers for the programming languages Lua and SQL. The results of our work led to a publication that we submitted at the end of 2019 (and which has been accepted by now). Besides the work on our program generator, we also began working on a test case reduction technique. It reduces the size of a randomly generated test program that triggers a compiler bug since this eases the search for the bug's root cause.

In 2020, we focussed on language-agnostic techniques for the automatic reduction of test programs. The scientific literature has proposed different reduction techniques, but since there is no conclusive comparison of these techniques yet, it is still unclear how efficient and effective the proposed techniques really are. We identified two main reasons for this, which also hamper the development and evaluation of new techniques. Firstly, the available implementations of the proposed reduction techniques use different implementation languages, program representations and input grammars. Therefore, a fair comparison of the proposed techniques is almost impossible with the available implementations. Secondly, there is no collection of (still unreduced) test programs that can be used for the evaluation of reduction techniques. As a result, the published techniques have only been evaluated with few test programs each, which compromises the significance of the published results. Furthermore, since some techniques have only been evaluated with test programs in a single programming language, it is still unclear how well these techniques generalize to other programming languages (i.e., how language-agnostic they really are). To close these gaps, we initiated the development of a framework that contains implementations of the most important reduction techniques and that enables a fair comparison of these techniques. In addition, we also started to work on a benchmark that already contains about 300 test programs in C and SMT-LIB 2 that trigger about 100 different bugs in real compilers. This benchmark not only enables conclusive comparisons of reduction techniques but also reduces the work for the evaluation of future techniques. Some first experiments already exposed that there is no reduction technique yet that performs best in all cases.

In this year, we also investigated how the random program generator that has been developed in the context of this research project can be extended to not only detect functional bugs but also performance problems in compilers. A new technique has been developed within a thesis that first generates a set of random test programs and then applies an optimization technique to gradually mutate these programs. The goal is to find programs for which the compiler under test has a considerably higher runtime than a reference implementation. First experiments have shown that this approach can indeed detect performance problems in compilers.

In 2021, we finished the implementation of the most important test case reduction techniques from the scientific literature as well as the construction of a benchmark for their evaluation. Building upon our framework and benchmark, we also conducted a quantitative comparison of the different techniques; to the best of our knowledge, this is by far the most extensive and conclusive comparison of the available reduction techniques to date. Our results show that there is no reduction technique yet that performs best in all cases. Furthermore, we detected that there are possible outliers for each technique, both in terms of efficiency (i.e., how quickly a reduction technique is able to reduce an input program) and effectiveness (i.e., how small the result of a reduction technique is). This indicates that there is still room for future work on test case reduction, and our results give some insights for the development of such future techniques. For example, we found that the hoisting of nodes in a program's syntax tree is mandatory for the generation of small results (i.e., to achieve a high effectiveness) and that an efficient procedure for handling list structures in the syntax tree is necessary. The results of our work led to a publication submitted and accepted in 2021.

In this year, we also investigated if and how the effectiveness of our program generator can be increased by considering the coverage of the input grammar during the generation. To this end and within a thesis, several context-free coverage metrics from the scientific literature have been adapted, implemented and evaluated. The results showed that the correlation between the coverage w.r.t. a context-free coverage metric and the ability to detect bugs in a compiler is rather limited. Therefore, more advanced coverage metrics that also consider context-sensitive, semantic properties should be evaluated in future work.

In 2022, we initiated the development of a new framework for the implementation of language-adapted reduction techniques. This framework introduces a novel domain-specific language (DSL) that allows the specification of reduction techniques in a simple and concise way. The framework and the developed DSL make is possible to easily adapt existing reduction techniques to the peculiarities and requirements of a specific programming language. It is our hope that such language-adapted reduction techniques can be even more efficient and effective than the existing, language-agnostic reduction techniques. In addition, the developed framework should also reduce the effort for the development of future reduction techniques; this way, our framework could make a valuable contribution to the research in this area.

In 2023, the focus of the research project was on list structures, which had already been briefly addressed in 2021: Almost all methods investigated since 2021 group nodes in the syntax tree into lists in order to select only the necessary nodes from these lists using a list reduction. Our experiments have shown that in some cases 70% or more of the reduction time is spent on lists with more than 2 elements. These lists are relevant because there are several list reduction methods in the scientific literature, but they do not differ for lists with 2 or fewer elements. Since they take such a large fraction of time, we have worked on integrating these different list reduction methods into our implementations of the major reduction methods developed in 2020/2021. In addition to the methods found in the literature, we also considered methods that are only described on a website or whose source code is freely accessible.

We also investigated how a list reduction can be interrupted at one point and resumed later. The idea was to reduce another list in the meantime, based on a prioritization, so that the list with the greater impact on the reduction always comes first. In some cases, the hoped-for speedup occurred, but questions remain that require further experiments with prioritizing reducers and interrupted list reduction methods.

In 2024, we successfully published the first results from the list reduction study: Replacing the list reductions can accelerate established reduction techniques by up to 74.7%. As expected, techniques that generate long lists benefit most from the change. We also found that the order of the list elements can save up to 44.1% of the runtime. But two aspects reduce the effectiveness of reordering:

1. The textual order in which the list elements are usually lined up is already quite a good order.
2. The same aspects that make a list procedure fast make it less sensitive to the order.

In two final theses we investigated two more aspects:

1. The tool developed from 2018 - 2021 for generating test programs uses the compiler under test only as a so-called “black box”, i.e., it generates programs without accessing any information from the tested compiler. The thesis used coverage information from the tested compiler to improve the generated programs.
2. Caching the results of the reductions saves time, as the compiler under test does not need to re-execute reduction candidates. However, naive implementations of these caches become very large. In 2023, a special caching method was introduced that can reduce the size of the cache by about 90%. The thesis dealt with the fact that unfortunately the original caching method was not suitable for all the reduction methods in our framework.

3.2 ORKA-HPC – *OpenMP for reconfigurable heterogenous architectures*

High-Performance Computing (HPC) is an important component of Europe’s capacity for innovation and it is also seen as a building block of the digitization of the European industry. Reconfigurable technologies such as Field Programmable Gate Array (FPGA) modules are gaining in importance due to their energy efficiency, performance, and flexibility.
There is also a trend towards heterogeneous systems with accelerators utilizing FPGAs. The great flexibility of FPGAs allows for a large class of HPC applications to be realized with FPGAs. However, FPGA

programming has mainly been reserved for specialists as it is very time consuming. For that reason, the use of FPGAs in areas of scientific HPC is still rare today.
In the HPC environment, there are various programming models for heterogeneous systems offering certain types of accelerators. Common models include OpenCL (http://www.opencl.org), OpenACC (https://www.openacc.org) and OpenMP (https://www.OpenMP.org). These standards, however, are not yet available for the use with FPGAs.

Goals of the ORKA project are:

1. Development of an OpenMP 4.0 compiler targeting heterogeneous computing platforms with FPGA accelerators in order to simplify the usage of such systems.
2. Design and implementation of a source-to-source framework transforming C/C++ code with OpenMP 4.0 directives into executable programs utilizing both the host CPU and an FPGA.
3. Utilization (and improvement) of existing algorithms mapping program code to FPGA hardware.
4. Development of new (possibly heuristic) methods to optimize programs for inherently parallel architectures.

In 2018, the following important contributions were made:

- Development of a source-to-source compiler prototype for the rewriting of OpenMP C source code (cf. goal 2).
- Development of an HLS compiler prototype capable of translating C code into hardware. This prototype later served as starting point for the work towards the goals 3 and 4.
- Development of several experimental FPGA infrastructures for the execution of accelerator cores (necessary for the goals 1 and 2).

In 2019, the following significant contributions were achieved:

- Publication of two peer-reviewed papers: “OpenMP on FPGAs - A Survey” and “OpenMP to FPGA Offloading Prototype using OpenCL SDK”.
- Improvement of the source-to-source compiler in order to properly support OpenMP-target-outlining for FPGA targets (incl. smoke tests).
- Completion of the first working ORKA-HPC prototype supporting a complete OpenMP-to-FPGA flow.
- Formulation of a genome for the pragma-based genetic optimization of the high-level synthesis step during the ORKA-HPC flow.
- Extension of the TaPaSCo composer to allow for hardware synchronization primitives inside of TaPaSCo systems.

In 2020, the following significant contributions were achieved:

- Improvement of the Genetic Optimization.
- Engineering of a Docker container for reliable reproduction of results.
- Integration of software components from project partners.
- Development of a plugin architecture for Low-Level-Platforms.
- Implementation and integration of two LLP plugin components.
- Broadening of the accepted subset of OpenMP.
- Enhancement of the test suite.

In 2021, the following significant contributions were achieved:

- Enhancement of the benchmark suite.
- Enhancement of the test suite.
- Successful project completion with live demo for the project sponsor.
- Publication of the paper "ORKA-HPC - Practical OpenMP for FPGAs".
- Release of the source code and the reproduction package.
- Enhancement of the accepted OpenMP subset with new clauses to control the FPGA related transformations.
- Improvement of the Genetic Optimization.
- Comparison of the estimated performance data given by the HLS and the real performance.
- Synthesis of a linear regression model for performance prediction based on that comparison.
- Implementation of an infrastructure for the translation of OpenMP reduction clauses.
- Automated translation of the OpenMP pragma "parallel for" into a parallel FPGA system.

In 2022, the following significant contributions were achieved:

- Generation and publication of an extensive dataset on HLS area estimates and actual performance.
- Creation and comparative evaluation of different regression models to predict actual system performance from early (area) estimates.
- Evaluation of the area estimates generated by the HLS.
- Publication of the paper "Reducing OpenMP to FPGA Round-trip Times with Predictive Modelling".
- Development of a method to detect and remove redundant read operations in FPGA stencil codes based on the polyhedral model.
- Implementation of the method for ORKA-HPC.
- Quantitative evaluation of that method to show the strength of the method and to show when to use it.
- Publication of the paper "Employing Polyhedral Methods to Reduce Data Movement in FPGA Stencil Codes".

In 2023, the following significant contributions were achieved:

- Development and implementation of an optimization method for canonical loop shells (e.g. from OpenMP target regions) for FPGA hardware generation using HLS. The core of the method is a loop restructuring based on the polyhedral model that uses loop tiling, pipeline processing, and port widening to avoid unnecessary data transfers from/to the onboard RAM of the FPGA, increase the number of parallel active circuits, maximize data throughput to FPGA board RAM, and hide read/write latencies.
- Quantitative evaluation of the strengths and application areas of this optimization method using ORKA-HPC.
- Publication of the method in the conference paper "Employing polyhedral methods to optimize stencils on FPGAs with stencil-specific caches, data reuse, and wide data bursts".
- Publication of a reproduction package for the optimization method.
- Presentation of the method at the conference "14th International Workshop on Polyhedral Compilation Techniques" in a half-hour talk.
- Development of a method for the fully automatic integration of multi-purpose caches into FPGA solutions generated from OpenMP.
- Evaluation of multi-purpose caches in combination with HLS generated hardware blocks.

- Publication of the paper “Multipurpose Cacheing to Accelerate OpenMP Target Regions on FPGAs” (Best Paper Award).

In 2024, the following significant contributions were achieved:

- Adaptation of several already published cacheing approaches to offloaded OpenMP codes and integration of the methods into ORKA-HPC.
- Development and evaluation of novel multi-layer caches for HLS kernels.
- Publication of the results in the publication “Multilayer Multipurpose Caches for OpenMP Target Regions on FPGAs” and presentation of the work at IWOMP 2024 in Perth.

3.3 SoftWater – *Software Watermarking*

Software watermarking means hiding selected features in code, in order to identify it or prove its authenticity. This is useful for fighting software piracy, but also for checking the correct distribution of open-source software (like for instance projects under the GNU license). The previously proposed methods assume that the watermark can be introduced at the time of software development, and require the understanding and input of the author for the embedding process. The goal of our research is the development of a watermarking framework that automates this process by introducing the watermark during the compilation phase into newly developed or even into legacy code. As a first approach we studied a method that is based on symbolic execution and function synthesis.
In 2018, two bachelor theses analyzed two methods of symbolic execution and function synthesis in order to determine the most appropriate one for our approach.
In 2019, we investigated the idea to use concolic execution in the context of the LLVM compiler infrastructure in order to hide a watermark in an unused register. Using a modified register allocation, one register can be reserved for storing the watermark.
In 2020, we extended the framework (now called LLWM) for automatically embedding software watermarks into source code (based on the LLVM compiler infrastructure) with further dynamic methods. The newly introduced methods rely on replacing/hiding jump targets and on call graph modifications.
In 2021, we added other adapted, dynamic methods that have already been published, as well as a newly developed method to LLWM. The added methods are based, among other things, on the conversion of conditional constructs into semantically equivalent loops or on the integration of hash functions, that leave the functionality of the program unchanged but increase its resilience. Our newly developed method IR-Mark now not only specifically selects the functions in which the code generator avoids using a certain register. IR-Mark now adds some dynamic computation of fake values that makes use of this register to blurr what is going on. There is a publication on both LLWM and IR-Mark.
In 2022, we added another adapted procedure to the LLWM framework. The method uses exception handling to hide the watermark.
In 2023, we adapted more methods to expand the LLWM framework. These include embedding techniques based on principles of number theory and aliasing.
In 2024, we developed three new watermarking techniques: *Register Expansion*, *SemaCall*, and *SideData*.
They construct hash-like arithmetics that generate a watermarking message from a secret key.
The first two techniques have been published in the paper “Register Expansion and SemaCall: 2 Low-overhead Dynamic Watermarks Suitable for Automation in LLVM” in the proceedings of the CheckMATE’24 workshop in Salt Lake City. We wrote an extended version containing the SideData watermark, currently under peer review for the DTRAP journal.

4 Teaching

The Chair for Programming Systems teaches the two compulsory modules *Algorithms and Data Structures (AuD)* and *Parallel and Functional Programming (PFP)* during the winter term. Due to changes of the examination regulations the lecture of AuD took place during the winter term 2021/22 for the last time, while the accompanying exercises continued to run until the winter term 2023/24. Since both modules are offered to many degree programs from different faculties (Computer Science, Information and Communication Technology, Mathematics, and many more), the numbers of attending students and examinations once again were high: 264 resp. 259 students attended PFP during the winter term 2023/24 resp. winter term 2024/25 – the number of examinations hit 49 in AuD and 548 in PFP (each during winter term 2023/24, summer term 2024, and winter term 2024/25). The Chair offers different modules on *Compiler Construction* and *Testing of Software Systems* to students specializing on programming systems. The seminars *Hallo Welt! für Fortgeschrittene* and *Machine Learning* were also fully booked within a short time.
The Chair for Programming Systems supervised eight master's thesis and six bachelor's thesis in total during the period under report.

ICPC – *International Collegiate Programming Contest an der FAU*: Since 1977 the International Collegiate Programming Contest (ICPC) takes place every year. Teams of three students try to solve about 13 programming problems within five hours. What makes this task even harder, is that there is only one computer available per team. The problems demand for solid knowledge of algorithms from all areas of computer science and mathematics, e.g., graphs, combinatorics, strings, algebra, and geometry. To solve the problems, the teams need to find a correct and efficient algorithm and implement it.
The ICPC consists of three rounds. First, each participating university hosts a local contest to find the up to three teams that are afterwards competing in one of the various regional contests. Germany lies in the catchment area of the Northwestern European Regional Contest (NWERC) with competing teams from Great Britain, Benelux, Scandinavia, etc. The winners of all regionals in the world (and some second place holders) advance to the world finals in spring of the following year.
On January 27, 2024, the Winter Contest took place once again. 108 teams from 17 universities participated, including 15 teams from Erlangen. Our best team finished 32nd. On June 22, the German Collegiate Programming Contest was held at several German universities, with 15 teams from Erlangen. The best FAU team secured the 22nd position out of 94 participating teams from all over Germany. The NWERC took place on November 24 in Delft. FAU was represented by one team, which finished in the 19th position among 80 participating teams. As usual, we also conducted the main seminar "Hello World! - Advanced Programming" in 2024.

5 Publications 2024

[1] Julian Brandner, Florian Mayer, and Michael Philippsen. Multilayer Multipurpose Caches for OpenMP Target Regions on FPGAs. In A. Espinosa, M. Klemm, B.R. de Supinski, M. Cytowski, and J. Klinkenberg, editors, *OpenMP: Advancing OpenMP for Future Accelerators*, volume 15195 of *Springer's Lecture Notes in Computer Science (LNCS)*, page 79–93, Cham, 2024. Springer. doi:10.1007/978-3-031-72567-8_6.

[2] Julian Brandner, Florian Mayer, and Michael Philippsen. Multilayer Multipurpose Caches for OpenMP Target Regions on FPGAs [Data set], 2024. doi:10.5281/zenodo.12755510.

[3] Leon Brasseler, Maximilian Stahlke, Thomas Robert Altstidl, Tobias Feigl, and Christopher Mutschler. Non-Line-of-Sight Detection for Radio Localization using Deep State Space Models. In

The fourteenth International Conference on Indoor Positioning and Indoor Navigation 2024 (IPIN 2024), pages 1–6, 2024.

[4] David Franco Contreras, Inigo Cortes, Georgios Kontes, Tobias Feigl, Christopher Mutschler, and Alexander Ruegamer. Reinforcement Learning Framework for Robust Navigation in GNSS Receivers. In *Proceedings of the 37th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2024)*, pages 2392–2408, 2024. doi:10.33012/2024.19853.

[5] Tobias Heineken and Michael Philippsen. Replication Package for 'The Impact of List Reduction for Language Agnostic Test Case Reducers', 2024. doi:10.5281/zenodo.13835515.

[6] Lucas Heublein, Felix Ott, and Tobias Feigl. Research Avenues for GNSS Interference Classification Robustness: Domain Adaptation, Continual Learning & Federated Learning. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, pages 1–4, 2024.

[7] Lucas Heublein, Nisha Lakshmana Raichur, Tobias Feigl, Tobias Brieger, Fin Heuer, Lennart Asbach, Jonathan Hansen, Alexander Ruegamer, and Felix Ott. Evaluation of (Un-)Supervised Machine Learning Methods for GNSS Interference Classification with Real-World Data Discrepancies. In *Proc. Intl. Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+) 2024*, volume 1, pages 1–18, 2024.

[8] Lucas Heublein, Nisha Lakshmana Raichur, Tobias Feigl, Tobias Brieger, Fin Heuer, Lennart Asbach, Alexander Ruegamer, and Felix Ott. GNSS Interference Monitoring: Resilience of Machine Learning Methods on Public Real-World Datasets. *Navigation, Journal of the Institute of Navigation*, 7:1–20, 2024.

[9] Marius Kamp. *Detecting Unrealizable Bit Vector Program Synthesis Problems*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2024. URL: https://open.fau.de/handle/openfau/33619, doi:10.25593/978-3-96147-794-4.

[10] Marius Kamp. Replication Package for "Detecting Unrealizable Bit Vector Program Synthesis Problems", Jan 2024. doi:10.5281/zenodo.10593916.

[11] Florian Mayer, Julian Brandner, and Michael Philippsen. Employing Polyhedral Methods to Optimize Stencils on FPGAs with Stencil-specific Caches, Data Reuse, and Wide Data Bursts. In *14th International Workshop on Polyhedral Compilation Techniques, (IMPACT 2024, in conjunction with HiPEAC 2024)*, page 12p, Jan 2024. URL: https://impact-workshop.org/impact2024/#mayer24-fpgas, doi:10.48550/arXiv.2401.13645.

[12] Florian Mayer, Julian Brandner, and Michael Philippsen. Employing polyhedral methods to optimize stencils on FPGAs with stencil-specific caches, data reuse, and wide data bursts [Reproduction Package], Jan 2024. doi:10.5281/zenodo.10396084.

[13] Felix Ott, Lucas Heublein, Nisha Lakshmana Raichur, Tobias Feigl, Jonathan Hansen, Alexander Ruegamer, and Christopher Mutschler. Few-Shot Learning with Uncertainty-based Quadruplet Selection for Interference Classification in GNSS Data. In *International Conference on Localization and GNSS*, page 7, 2024. doi:10.1109/ICL-GNSS60721.2024.10578525.

[14] Jonathan Ott, Jonas Pirkl, Maximilian Stahlke, Tobias Feigl, and Christopher Mutschler. Radio Foundation Models: Pre-training Transformers for 5G-based Indoor Localization. In *The fourteenth edition of the International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–6, 2024.

[15] David Schwarzbeck. Erweiterung eines Rahmenprogramms für das automatische Einfügen von Software- Wasserzeichen in Quellcode. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2024. URL: https://cs2-gitlab.cs.fau.de/softwater/ma-david/-/raw/master/thesis/thesis.pdf?ref_type=heads.

[16] David Schwarzbeck, Daniela Novac, and Michael Philippsen. Register Expansion and SemaCall: 2 Low-overhead Dynamic Watermarks Suitable for Automation in LLVM. In *CheckMATE '24: Proceedings of the 2024 Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks*, pages 1–10, New York, 2024. ACM. URL: https://dl.acm.org/doi/10.1145/3689934.3690815#, doi:10.1145/3689934.3690815.

[17] David Schwarzbeck, Jan Schuh, Mercel Hammrich, Michael Philippsen, and Daniela Novac. Register Expansion and SemaCall: 2 low-overhead dynamic Watermarks suitable for Automation in LLVM [Source code and Raw Experiment data], 2024. doi:10.5281/zenodo.13337275.

[18] Maximilian Stahlke, Tobias Feigl, Sebastian Kram, Björn Eskofier, and Christopher Mutschler. Uncertainty-Based Fingerprinting Model Monitoring for Radio Localization. *IEEE Journal of Indoor and Seamless Positioning and Navigation*, 2024.

[19] Johannes Rossouw Van Der Merwe, David Franco Contreras, Tobias Feigl, and Alexander Ruegamer. Optimal machine learning and signal processing synergies for low-resource GNSS interference detection and classification. *IEEE Transactions on Aerospace and Electronic Systems*, pages 3–17, Jan 2024. doi:10.1109/TAES.2023.3349360.