2023 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., Dr.-Ing. Norbert Oster, Akad. ORat, Prof. Dr. Michael Philippsen (Ordinarius), Prof. em. Dr. Hans Jürgen Schneider (Emeritus), Dipl.-Ing. Frank Deserno (IT-support, until March 31, 2023), Bernd Bierlein (IT-Support, from May 1, 2023 until October 31, 2023), Margit Zenk (Secretary).

Guests and external teaching staff: Dr.-Ing. Veronika Dashuber, Dr.-Ing. Klaudia Dussa-Zieger, Dr.-Ing. Tobias Feigl, Dr.-Ing. Martin Jung, 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 AnaCoRe – *Analysis of Code Repositories*

Software developers often modify their projects in a similar or repetitive way. The reasons for these changes include the adoption of a changed interface to a library, the correction of mistakes in functionally similar components, or the parallelization of sequential parts of a program. If developers have to perform the necessary changes on their own, the modifications can easily introduce errors, for example due to a missed change location. Therefore, an automatic technique is desireable that identifies similar changes and uses this knowledge to support developers with further modifications.

Extraction of Code-Changes
In 2017, we developed a new code recommendation tool called ARES (Accurate REcommendation System). It creates more accurate recommendation compared to previous tools as its algorithms take care of code movements during pattern and recommendation creation. The foundation of ARES lies in the

comparison of two versions of the same program. It extracts the changes between the two versions and creates patterns based on the changed methods. ARES uses these patterns to suggest similar changes for the source code of different programs automatically.
The extraction of code changes is based on trees. In 2016 we developed (and visibly published) a new tree-based algorithm (MTDIFF) that improves the accuracy of the change extraction.

Symbolic Execution of Code-Fragments
In 2014 we developed a new symbolic code execution engine called SYFEX to determine the behavioral similarity of two code fragments. In this way we aim to improve the quality of the recommendations. Depending on the number and the generality of the patterns in the database, it is possible that without the new engine SIFE generates some unfitting recommendations. To present only the fitting recommendations to the developers, we compare the summary of the semantics/behavior of the recommendation with summary of the semantics/behavior of the database pattern. If both differ too severely, our tool drops the recommendation from the results. The distinctive features of SYFEX are its applicability to isolated code fragments and its automatic configuration that does not require any human interaction.
In 2015 SYFEX was refined and applied to code fragments from the repositories of different software projects. In 2016 we investigated to which extend SYFEX can be used to gauge the semantic similarity of submissions for a programming contest. In 2017 and 2018 we optimized the implementation of SYFEX. We also began collecting a data set of semantically similar methods from open source repositories. We published this data set in 2019.
Techniques for symbolic execution use algorithms to check the satisfiability of logical/mathematical expressions in order to detect valid execution paths in a program. Usually, these algorithms account for a large part of the total runtime of a symbolic execution. To accelerate this satisfiability check, we experimented with a technique to replace complicated expressions with simpler equivalent expressions. These simpler expressions are obtained by using program synthesis. In the year 2020, we extended this program synthesis with a novel technique that can quickly detect whether a fixed set of operations can be used to construct an expression that is equivalent to the complicated expression. We published this approach in 2021 and were able to show that the technique can reduce the runtime of common program synthesizers by 33% on average. We subsequently extended this technique to other classes of program synthesis problems. In 2022, we performed a comprehensive evaluation of these extensions. This evaluation showed that these extensions similarly improve the runtime of program synthesizers on a larger class of program synthesis problems. We completed the work on unrealizability detectors for bit vector program synthesis in 2023 and described it in detail in a Dissertation.

Detection of Semantically Similar Code Fragments
SYFEX computes the semantic similarity of two code fragments. Therefore, it allows to identify pairs or groups of semantically similar code fragments (semantic clones). However, the high runtime of SYFEX (and similar tools) limit their applicability to larger software projects. In 2016, we started the development of a technique to accelerate the detection of semantically similar code fragments. The technique is based on so-called base comparators that compare two code fragments using a single criterion (e.g., the number of used control structures or the structure of the control flow graph) and that have a low runtime. These base comparators can be combined to form a hierarchy of comparators. To compute the semantic similarity of two code fragments as accurately as possible, we use genetic programming to search for hierarchies that approximate the similarity values as reported by SYFEX for a number of pairs of code fragments. A prototype implementation confirmed that the method is capable of detecting pairs of semantically similar code fragments.
We further improved the implementation of this approach in 2017 and 2018. Additionally, we focused on evaluating the approach with pairs of methods from software repositories and from programming exercises. Moreover, we created a data set of semantically similar methods from open-source software repositories that we published in 2019.

Techniques for symbolic execution rely on algorithms to detect the satisfiability of logic/mathematic expressions. These are used to detect whether an execution path in a program is feasible. The algorithms often use a large amount of the total computation time. To improve the speed of this satisfiability check, in the years 2019 and 2020 we experimented with a technique to replace complicated expressions with simpler expressions that have the same meaning. These simpler expressions result from the application of program synthesis. In 2020 we augmented the program synthesis with a novel approach to detect beforehand if some operations can form an expression with the same meaning as a more complicated expression.

Semantic Code Search

The functionality that has to be implemented during the development of a software product is often already available as part of program libraries. It is often advisable to reuse such an implementation instead of rewriting it, for example to reduce the effort for developing and testing the code.

To reuse an implementation that fits the purpose, developers have to find it first. To this end developers already use code search engines on a regular basis. State-of-the-art search engines work on a syntactic level, i.e., the user specifies some keywords or names of variables and methods that should be searched for. However, current approaches do not consider the semantics of the code that the user seeks. As a consequence, relevant but syntactically different implementations often remain undetected ("false negatives") or the results include syntactically similar but semantically irrelevant implementations ("false positives"). The search for code fragments on a semantic level is the subject of current research.

In 2017 we began the development of a new method for semantic code search. The user specifies the desired functionality in terms of input/output examples. A function synthesis algorithm from the literature is then used to create a method that implements the specified functionality as accurately as possible. Using our approach to detect similar code fragments, this synthesized method is then compared to the methods of program libraries to find semantically similar implementations. These implementations are then presented as search results to the user. A first evaluation of our prototypical implementation shows the feasibility and practicability of the approach.

Clustering of Similar Code-Changes

To create generalized change patterns, it is necessary that the set of extracted code changes is split into subsets of changes that are similar to each other. In 2015 this detection of similar code changes was improved and resulted in a new tool, called C3. We developed and evaluated different metrics for a pairwise similarity comparison of the extracted code changes. Subsequently, we evaluated different clustering algorithms known from the literature and implemented new heuristics to automatically choose the respective parameters to replace the previous naive approach for the detection of similar code changes. This clearly improved the results compared to the previous approach, i.e., C3's new techniques detect more groups of similar changes that can be processed by SIFE to generate recommendations.

The aim of the second improvement is to automatically refine the resulting groups of similar code changes. For this purpose we evaluated several machine learning algorithms for outlier detection to remove those code changes that have been spuriously assigned to a group.

In 2016 we implemented a new similarity metric for the comparison of two code changes that essentially considers the textual difference between the changes (as generated, for example, by the Unix tool 'diff'). We published both a paper on C3 and the dataset (consisting of groups of similar changes) that we generated for the evaluation of our tool under an open-source license, see https://github.com/FAU-Inf2/cthree. This dataset can be used as a reference or as input data for future research. In addition, we prototypically extended C3 by techniques for an incremental similarity computation and clustering. This allows us to reuse results from previous runs and to only perform the absolutely necessary work whenever new code changes are added to a software archive.

3.2 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.

3.3 Holoware – *Cooperative Exploration and Analysis of Software in a Virtual/Augmented Reality Appliance*

Understanding software has a large share in the programming efforts of a software systems, up to 30% in development projects and up to 80% in maintenance projects. Therefore, an efficient and effective way for comprehending software is necessary in a modern software engineering workplace. Three-dimensional software visualization already boosts comprehension and efficiency, so utilization of the latest virtual reality techniques seems natural. Within the scope of the Holoware project, we create an environment to cooperatively explore and analyze a software project using virtual/augmented reality techniques as well as artificial intelligence algorithms. The software project in question is being visualized in said virtual reality, such that multiple participants can simultaneously explore and analyze the software. They can cooperate by communicating about their findings. Different participants benefit from different perspectives on the software, which is augmented by domain specific additional information. This provides them with intuitive access to the structure and behaviour of the software. Various use cases are possible, for

example the cooperative analysis of a run time anomaly in a team of domain experts. The domain experts can see the same static structure, augmented with domain specific and detailed information. In the VR environment, they can share their findings and cooperate using their different expertise.

In addition, the static and dynamic properties of the software system are analyzed. Static properties include source code, static call relationships or metrics such as LoC, cyclomatic complexity, etc. Dynamic properties can be grouped into logs, traces, runtime metrics, or configurations that are read in at runtime. The challenge lies in aggregating, analyzing, and correlating this wealth of information. An anomaly and significance detection is developed that automatically detects both structural and runtime anomalies. In addition, a prediction system is set up to make statements about component health. This makes it possible, for example, to predict which components are at risk of failing in the near future. Previously, the log entries were added to the traces, creating a detailed picture of the dynamic call relationships. These dynamic relationships are mapped to the static call graph because they describe calls that do not result from the static analysis (for example, REST calls across several distributed components).

In 2018, the following significant contributions have been made:

- Development of a functional VR visualization prototype for demonstration and research purposes.
- Mapping between dynamic run time data and static structure (required by later analysis and visualization tasks).
- First draft and implementation of the trace anomaly detection by an unsupervised learning procedure. Evaluation and further improvements will follow in the coming months.

In 2019 we achieved the following improvements:

- Extension of the prototype to display dynamic software behaviour.
- Cooperative (remote-)usability of the visualization prototype.
- Interpretation of commit messages for anomaly detection.
- Clustering system calls according to use cases.

Our paper “Towards Collaborative and Dynamic Software Visualization in VR” has been accepted for publication at the International Conference on Computer Graphics Theory and Applications (VISIGRAPP) 2020. It presents the efficiency of our prototype at increasing the software understanding process. In 2020, our paper “A Layered Software City for Dependency Visualization” was accepted at the International Conference on Computer Graphics Theory and Applications (VISIGRAPP) 2021 and later received the “Best Paper Award”. We demonstrated that our Layered Layout for Software Cities simplifies the analysis of software architecture and outperforms the standard layout by far. We successfully concluded the research project with a final prototype and the resulting publications.

In 2021, after the end of the official project funding we were asked to submit an extended version of the award paper (“Static And Dynamic Dependency Visualization In A Layered Software City”) for review to a journal. Here we present a night view of the city that shows dynamic dependencies as arcs. We thus addressed a central, yet remaining issue: the visualization of dynamic dependencies. In the paper “Trace Visualization within the Software. City Metaphor: A Controlled Experiment on Program Comprehension” at the IEEE Working Conference on Software Visualization (VISSOFT), we displayed dynamic dependencies within the Software City by means of light intensities and were able to show that this representation is more helpful than drawing all dependencies. Also for this paper, we were invited to submit an extended journal article “Trace Visualization within the Software City Metaphor: Controlled Experiments on Program Comprehension” for review. This article demonstrates an extended visualization of dynamic dependencies and color arcs based on HTTP status codes.

In 2022, both journal papers were accepted: “Static And Dynamic Dependency Visualization in a Layered Software City” is published in Springer Nature Computer Science Journal and “Trace Visualization within the Software City Metaphor: Controlled Experiments on Program Comprehension” was accepted for the Information and Software Technology Journal. For the finalization of Holoware, all extensions were combined into one single visualization. For this purpose, different views were applied, allowing the user to switch between them: in the day view, the software architecture can be analyzed in the novel Holoware layered layout, and in the night view, dynamic dependencies are displayed. As part of a master thesis, Holoware was also implemented as an AR visualization, so that it can easily be used as a showcase or in everyday work.

In mid 2023, we finalized the project with the dissertation "Visualizing the statics, dynamics and infrastructure of software using the city metaphor". It summarizes all investigated aspects: (a) the static structure of the system to understand the software architecture, (b) the dynamics of the system to understand the dynamic dependencies (e.g. modern microservice architectures), and (c) the infrastructure of the system to analyze costs and promote the understanding of software operation. We also uncovered another use case: the use of Holoware at trade fairs. The visualization of the software makes it easy to get into conversation with other software developers, as the visualized software can be discussed immediately. To this end, we simplified the setup of the AR and VR visualization so that Holoware can easily be started without a lot of prior technical knowledge. In addition, we improved the contrast of the visualization to make it easier to recognize outlines and arcs, especially in very bright lighting conditions.

3.4 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).

3.5 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.

3.6 V&ViP – *Verification and validation in industrial practice*

Detection of flaky tests based on software version control data and test execution history

Regression tests are carried out often and because of their volume also fully automatically. They are intended to ensure that changes to individual components of a software system do not have any unexpected side effects on the behavior of subsystems that they should not affect. However, even if a test case executes only unmodified code, it can still sometimes succeed and sometimes fail. This so-called "flaky" behavior can have different reasons, including race conditions due to concurrent execution or temporarily unavailable resources (e.g., network or databases). Flaky tests are a nuisance to the testing process in every respect, because they slow down or even interrupt the entire test execution and they undermine the confidence in the test results: if a test run is successful, it cannot necessarily be concluded that the program is really error-free, and if the test fails, expensive resources may have to be invested to reproduce and possibly fix the problem.
The easiest way to detect test flakyness is to repeatedly run test cases on the identical code base until the test result changes or there is a reasonable statistical confidence that the test is non-flaky. However, this is rarely possible in an industrial environment, as integration or system tests can be extremely time-consuming and resource-demanding, e.g., because they require the availability of special test hardware. For this reason, it is desirable to classify test cases with regard to their flakyness without repeated re-execution, but instead to only use the information already available from the preceding development and test phases.
In 2022, we implemented and compare various so-called black box methods for detecting test flakyness and evaluated them in a real industrial test process with 200 test cases. We classify test cases exclusively on the basis of generally available information from version control systems and test execution tools, i.e., in particular without an extensive analysis of the code base and without monitoring of the test coverage, which would in most cases be impossible for embedded systems anyway. From the 122 available indicators (including the test execution time, the number of lines of code, or the number of changed lines of code in the last 3, 14, and 54 days) we extracted different subsets and examined their suitability for detecting test flakyness using different techniques. The methods applied on the feature subsets include rule-based methods (e.g., "a test is flaky if it has failed at least five times within the observation window, but not five times in a row"), empirical evaluations (including the computation of the cumulative weighted "flip rate", i.e., the frequency of alternating between test success and failure) as well as various methods from the domain of machine learning (e.g., classification trees, random forest, or multi-layer perceptrons). By using AI-based classifiers together with the SHAP approach for explaining AI models we determined

the four most important indicators (“features”) for detecting test flakyness in the industrial environment under consideration. The so-called “gradient boosting” with the complete set of indicators has proven to be optimal (with an F1-score of 96.5%). The same method with only four selected features achieved just marginally lower accuracy and recall values (with almost the same F1 score).

Synergies of a-priori and a-posteriori analysis methods to explain artificial intelligence

Artificial intelligence is rapidly conquering more domains of everyday life and machines make more critical decisions: braking or evasive maneuvers in autonomous driving, credit(un)worthiness of individuals or companies, diagnosis of diseases from various examination results (e.g., cancer detection from CT/MRT scans), and many more. In order for such a system to receive trust in a real-life productive setting, it must be ensured and proven that the learned decision rules are correct and reflect reality. The training of a machine model itself is a very resource-intensive process and the quality of the result can usually only be quantified afterwards with extremely great effort and well-founded specialist knowledge. The success and quality of the learned model not only depends on the choice of a particular AI method, but is also strongly influenced by the magnitude and quality of the training data.
In 2022, we therefore examined which qualitative and quantitative properties an input set must have (“a priori evaluation”) in order to achieve a good AI model (“a posteriori evaluation”). For this purpose, we compared various evaluation criteria from the literature and we defined four basic indicators based on them: representativeness, freedom from redundancy, completeness, and correctness. The associated metrics allow a quantitative evaluation of the training data in advance of preparing the model. To investigate the impact of poor training data on an AI model, we experimented with the so-called “dSprites” dataset, a popular generator for image files used in the evaluation of image resp. pattern recognition methods. This way, we generated different training data sets that differ in exactly one of the four basic indicators and have quantitatively different “a priori quality”. We used all of them to train two different AI models: Random Forest and Convolutional Neural Networks. Finally, we quantitatively evaluated the quality of the classification by the respective model using the usual statistical measures (accuracy, precision, recall, F1-score). In addition, we used SHAP (a method for explaining AI models) to determine the reasons for any misclassification in cases of poor data quality. As expected, the model quality highly correlates with the training data quality: the better the latter is with regard to the four basic indicators, the more precise is the classification of unknown data by the trained models. However, a noteworthy discovery has emerged while experimenting with the lack of redundancy: If a trained model is evaluated with completely new/unknown inputs, the accuracy of the classification is sometimes significantly worse than if the available input data is split into a training and an evaluation data set: In the latter case, the a posteriori evaluation of the trained AI system misleadingly suggests a higher model quality.

Few-Shot Out-of-Domain Detection in Natural Language Processing Applications

Natural language processing (NLP for short) using artificial intelligence has many areas of application, e.g., telephone or written dialogue systems (so-called chat bots) that provide cinema information, book a ticket, take sick leave, or answer various questions arising during certain industrial processes. Such chat bots are often also involved in social media, e.g., to recognize critical statements and to moderate them if necessary. With increasing progress in the field of artificial intelligence in general and NLP in particular, self-learning models are spreading that dynamically (and therefore mostly unsupervised) supplement their technical and linguistic knowledge from concrete practical use. But such approaches are susceptible to intentional or unintentional malicious disguise. Examples from industrial practice have shown that chat bots quickly “learn” for instance racist statements in social networks and then make dangerous extremist statements. It is therefore of central importance that NLP-based models are able to distinguish between valid “In-Domain (ID)” and invalid “Out-Of-Domain (OOD)” data (i.e., both inputs and outputs). However, the developers of an NLP system need an immense amount of ID and OOD training data for the initial training of the AI model. While the former are already difficult to find in sufficient quantities,

the a priori choice of the latter is usually hardly possible in a meaningful way.
In 2022, we therefore examined and compared different approaches to OOD detection that work with little to no training data at all (hence called "few-shot"). The currently best and most widespread, transformer-based and pre-trained language model RoBERTa served as the basis for the experimental evaluation. To improve the OOD detection, we applied "fine-tuning" and examined how reliable the adaptation of a pre-trained model to a specific domain can be done. In addition, we implemented various scoring methods and evaluated them to determine threshold values for the classification of ID and OOD data. To solve the problem of missing training data, we also evaluated a technique called "data augmentation": with little effort GPT3 ("Generative Pretrained Transformer 3", an autoregressive language model that uses deep learning to generate human-like text) can generate additional and safe ID and OOD data to train and evaluate NLP models.

Application of weighted combinatorics in the generation and selection of parameters and their representatives in software testing

Some functional testing methods (so-called black box tests), such as the equivalence class testing or boundary value analysis, focus on individual parameters. For these parameters, they determine representatives (values or classes of values) to be considered in the test. Since not just a single parameter but several parameters are usually required to perform such tests, representatives of several parameters must be combined with each other to be used for test execution. Well-understood combinatorial methods such as "All Combinations", "Pair-wise" or "Each choice" are usually used for this purpose. They do not take into account information about weights (attributes such as importance or priority) of the parameters and equivalence class representatives, which would affect the number of associated test cases (e.g. due to importance) or their recommended order (in terms of prioritization). In addition, in the case of the equivalence class method, there are scenarios in which a combination of several invalid classes in a single test case could optionally be explicitly desired, completely undesirable or limited to a certain number in order to specifically test fault combinations on the one hand, but also to simplify fault localization on the other. There is reason to believe that by considering such weights and options, more targeted and ultimately more efficient test cases can be derived.

In 2023, we evaluated and compared known combinatorial approaches that take into account weights when combining parameters or their values. Based on this, we developed a novel approach to generate and select parameters and their representatives in software testing. The proposed method uses a weighting system to prioritize the individual parameters, their equivalence classes and concrete representatives, in a set of test cases. If necessary, their interactions can also be specifically weighted in order to allow certain combinations to occur more frequently in the generated test cases. To evaluate the approach, we defined a suitable prototype data structure that represents the various weightings. We then implemented evaluation functions for existing sets of test cases in order to quantitatively determine how well such a test case set satisfies the specified combinatorics. In a further step, we used these evaluation functions in combination with various systematic methods and heuristics (SAT solver Z3, simulated annealing, and genetic algorithms) to generate new test cases that match the weighting or to optimize existing sets by adding missing test cases. Simulated Annealing was the fastest and gave the best results in the test series. Although the SAT-approach worked well for small problems, it was no longer practical for larger test cases due to exorbitant runtimes.

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. Because of the same reason the module PFP was last given in the summer term 2022 before it started again in the winter term 2023/24, with a shifted cycle. 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: 127 resp. 63 students attended AuD during the winter term 2022/23 resp. summer term 2023 – the number of examinations hit 117 resp. 58 in AuD and 120 resp. 64 in PFP. 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 four master's thesis and two 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 (2023 in Sharm El Sheikh, Egypt).
On January 28, 2023, the Winter Contest took place once again. 75 teams from 16 universities participated, including 13 teams from Erlangen. Our best team finished 10th. On June 17, the German Collegiate Programming Contest was held at several German universities, with 14 teams from Erlangen. The best FAU team secured the 11th position out of 105 participating teams from all over Germany. The NWERC took place on November 26 in Delft. FAU was represented by 3 teams, which finished on the 32nd, 96th, and 125th positions among 143 participating teams. As usual, we also conducted the main seminar "Hello World! - Advanced Programming" in 2023.

5 Publications 2023

[1] Julian Brandner, Florian Mayer, and Michael Philippsen. Multipurpose Cacheing to Accelerate OpenMP Target Regions on FPGAs [Data set], 2023. doi:10.5281/zenodo.8055889.

[2] Julian Brandner, Florian Mayer, and Michael Philippsen. Multipurpose Cacheing to Accelerate OpenMP Target Regions on FPGAs (Best Paper Award). In Simon McIntosh-Smith, Tom Deakin, Michael Klemm, Bronis R. de Supinski, and Jannis Klinkenberg, editors, *OpenMP: Advanced Task-Based, Device and Compiler Programming*, volume 14114 of *Springer's Lecture Notes in Computer Science (LNCS)*, pages 147–162, 2023. doi:10.1007/978-3-031-40744-4{_}10.

[3] Veronika Dashuber. *Visualisierung der Statik, Dynamik und Infrastruktur von Software mit Hilfe der Stadt-Metapher*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2023. URL: https://opus4.kobv.de/opus4-fau/files/23373/DissertationVeronikaDashuberPress.pdf.

[4] Tobias Feigl, Tobias Brieger, Felix Ott, Jonathan Hansen, David Franco Contreras, Alexander Ruegamer, and Wolfgang Felber. Evaluation of (Un-)Supervised Machine-Learning-Based Detection, Classification, and Localization Methods of GNSS Interference in the Real World. In *Proc. Intl. Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+)*, pages 1–13, Denver, CO, 2023.

[5] Martin Gruber, Michael Heine, Norbert Oster, Michael Philippsen, and Gordon Fraser. Practical Flaky Test Prediction using Common Code Evolution and Test History Data. In *Proc. 16th IEEE Intl. Conf. Software Testing, Verification and Validation, ICST 2023*, pages 210–221, Dublin, Ireland, 2023. doi:10.1109/ICST57152.2023.00028.

[6] Martin Gruber, Michael Heine, Norbert Oster, Michael Philippsen, and Gordon Fraser. Practical Flaky Test Prediction using Common Code Evolution and Test History Data [replication package], 2023. doi:10.6084/m9.figshare.21363075.

[7] Florian Mayer, Julian Brandner, and Michael Philippsen. Employing Polyhedral Methods to Reduce Data Movement in FPGA Stencil Codes. In Charith Mendis and Lawrence Rauchwerger, editors, *Proc. of the 35rd Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC 2022)*, volume 13829 of *Lecture Notes in Computer Science (LNCS)*, pages 47–63, Chicago, IL, 2023. doi:10.1007/978-3-031-31445-2_4.

[8] Felix Ott, Nisha Lakshmana Raichur, David Ruegamer, Tobias Feigl, Heiko Neumann, Bernd Bischl, and Christopher Mutschler. Benchmarking Visual-Inertial Deep Multimodal Fusion for Relative Pose Regression and Odometry-aided Absolute Pose Regression, 2023. doi:10.48550/arXiv.2208.00919.

[9] Jonathan Ott, Maximilian Stahlke, Sebastian Kram, Tobias Feigl, and Christopher Mutschler. Multipath Delay Estimation in Complex Environments using Transformer. In *Proc. 13th Intl. Conf. Indoor Positioning and Indoor Navigation (IPIN 2023)*, pages 1–6, Nuremberg, Germany, 2023. doi:10.1109/IPIN57070.2023.10332470.

[10] Maximilian Stahlke, Tobias Feigl, Sebastian Kram, Björn Eskofier, and Christopher Mutschler. Uncertainty-based Fingerprinting Model Selection for Radio Localization. In *Proc. 13th Intl. Conf. Indoor Positioning and Indoor Navigation (IPIN 2023)*, pages 1–6, Nuremberg, Germany, 2023. doi:10.1109/IPIN57070.2023.10332531.

[11] Maximilian Stahlke, Yammine George, Tobias Feigl, Björn Eskofier, and Christopher Mutschler. Velocity-Based Channel Charting with Spatial Distribution Map Matching. *IEEE Sensors Journal*, pages 1–8, 2023. doi:10.48550/arXiv.2311.08016.

[12] Maximilian Stahlke, George Yammine, Tobias Feigl, Björn Eskofier, and Christopher Mutschler. Indoor Localization with Robust Global Channel Charting: A Time-Distance-Based Approach. *IEEE Transactions on Machine Learning in Communications and Networking*, 1:3–17, 2023. doi:10.1109/TMLCN.2023.3256964.

[13] Johannes van der Merwe, David Contreras Franco, Jonathan Hansen, Tobias Brieger, Tobias Feigl, Felix Ott, Jdidi Dorsaf, Alexander Ruegamer, and Wolfgang Felber. Low-cost COTS GNSS interference monitoring, detection, and classification system. *Sensors*, 23(7):1–42, 2023. doi:10.3390/s23073452.