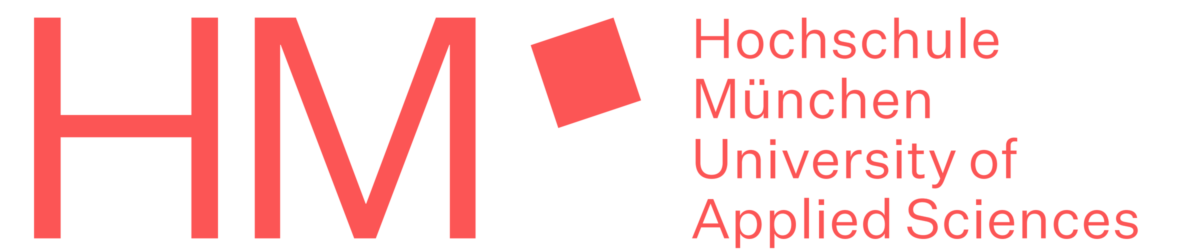
Sizalizer: Analysis Framework for ISA Optimization

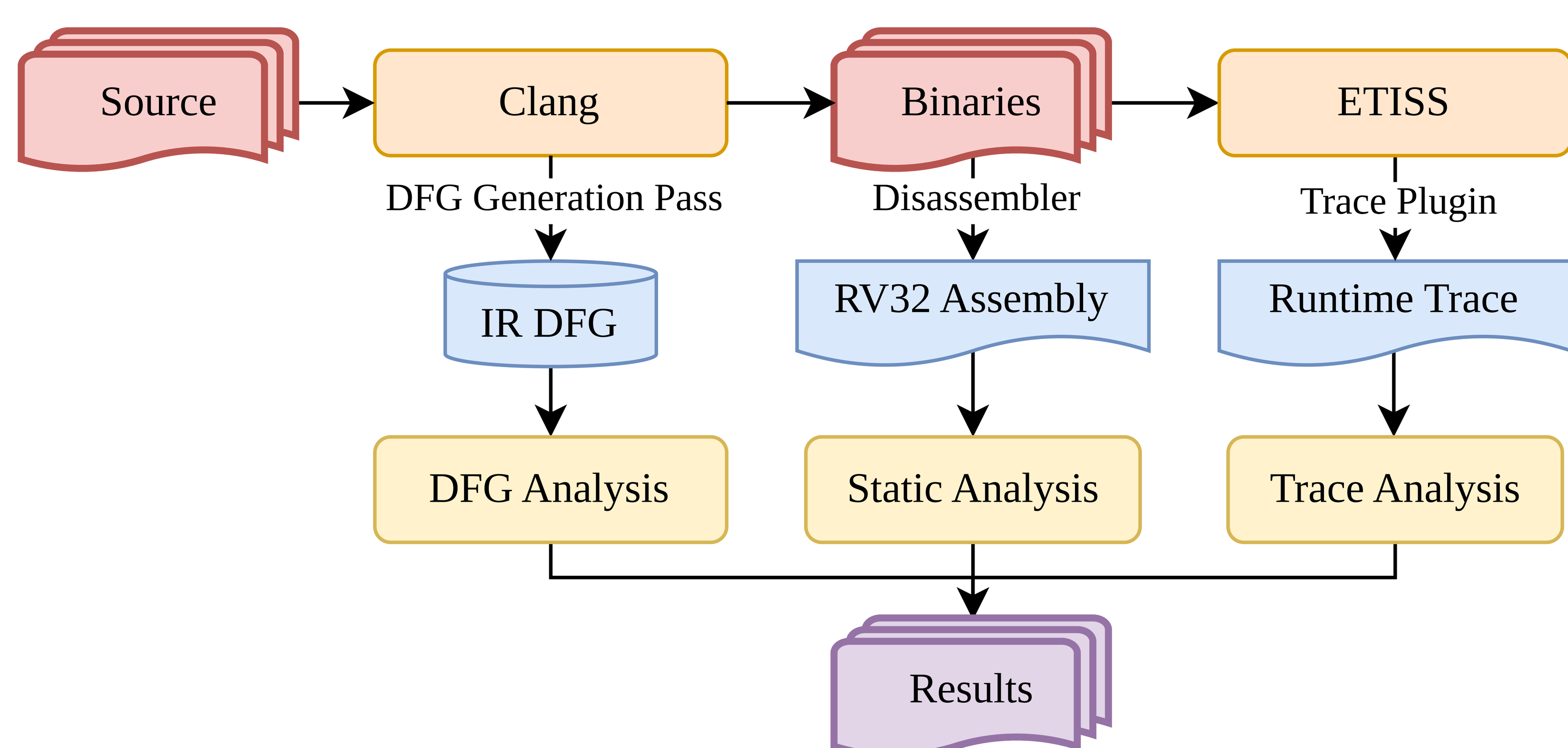
Andreas Hager-Clukas, Hochschule München
Computer Science and Mathematics
Supervisor: Prof. Stefan Wallentowitz



Motivation

Most microprocessors are produced for embedded systems [1]. The Information and Communication Technology (ICT) industry contributes between 1.8 % and 2.8 % to the Global Greenhouse Gas (GHG) emissions footprint [2], a statistic that is on an upward trajectory and is anticipated to maintain its climb [3]. To mitigate this upward trend, it's imperative to forge improved methodologies in both hardware and software engineering that match strides with the rapid progression of technology yet do not exacerbate GHC impacts. One promising route is the refinement of the Instruction Set Architecture (ISA), where enhancements can lead to existing programs being executable within a reduced memory footprint. This is particularly applicable for embedded microprocessors that are designated to operate a single application or only applications with domain-specific characteristics, paving the way for the utilization of smaller-scale processors a move that inherently cuts costs. Progressive evolution of current ISAs, like RV32, calls for a profound understanding of the system designer's software prerequisites. Alas, the gap remains in the lack of an analytical framework that scrutinizes and interprets software within the embedded sphere to facilitate constructive refinement of the ISA under examination.

Framework



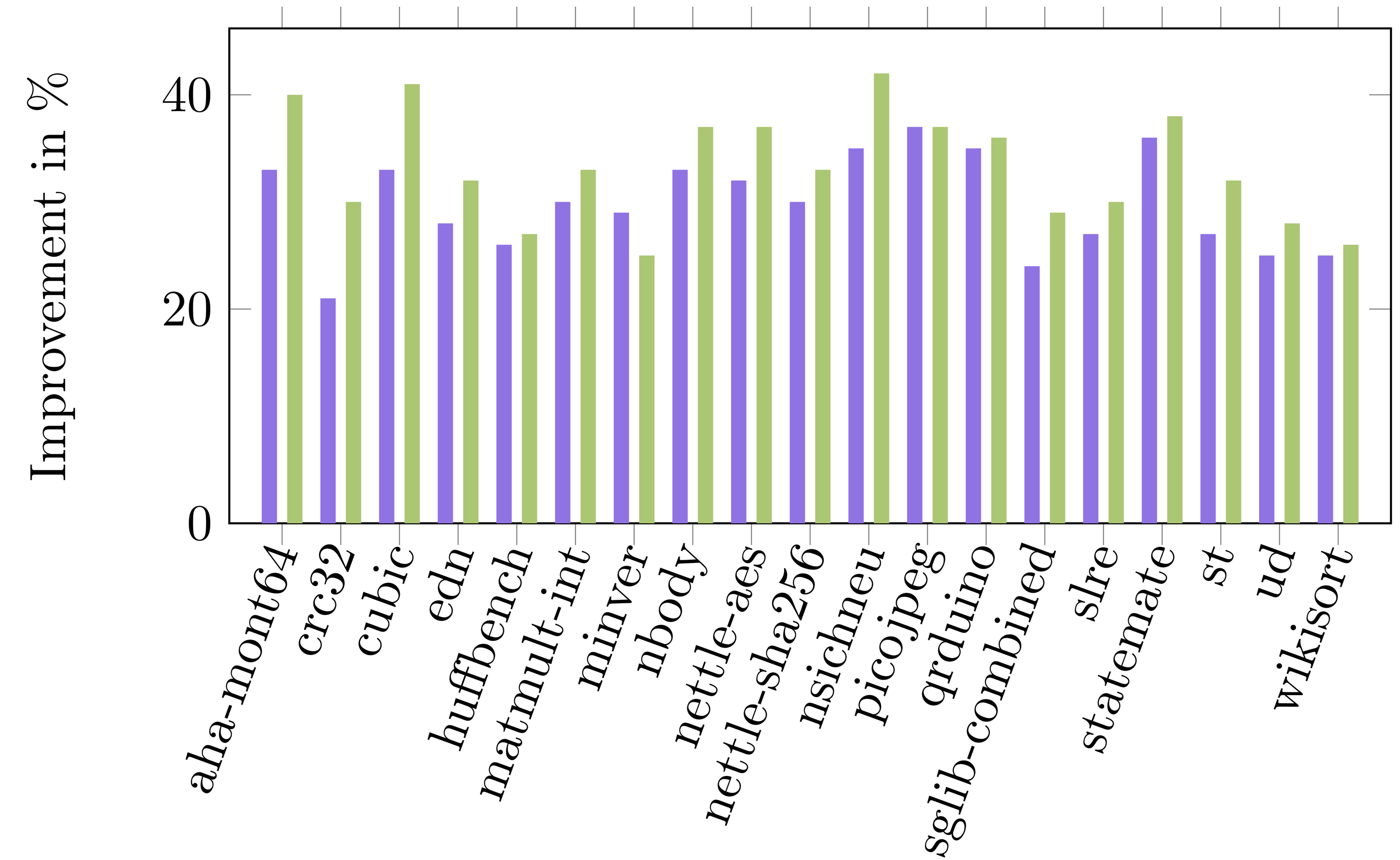
The architecture of Sizalizer is organized into a three-tiered analytical framework. This hierarchical model is visually represented in the Figure, which illustrates the extraction of relevant data at progressive stages. Source and build artifacts denoted in red, analysis modules in yellow, and data representations for permanence in blue.

The analytical process consists of three distinct layers. The uppermost layer involves the interprocedural data flow throughout the entire program. The corresponding Data Flow Graph (DFG) is synthesized from the LLVM Intermediate Representation (LLVM-IR) during an analysis pass while compiling the source with the C/C++ compiler, Clang. Because of its LLVM foundation, Clang is an ideal choice for enhancements. The well-defined nature of LLVM-IR serves as a robust foundation for subsequent analysis. The generated DFG is stored in the graph database, Memgraph, which is selected for its accessible and performant C++ interface [5]. Memgraph also has exceptional performance compared to other graph databases, such as Neo4j [4]. An external program interfaces with Memgraph to derive pertinent statistics and structures related to the analysis target. Furthermore, the use of the graph query language Cypher enables pattern and structure matching within the DFG, thereby extending the potential of static analysis through a standardized interface and interchangeable client.

The second analytical layer focuses on the static machine code level and operates on executable binary programs, or binaries. This analysis specializes in the RISC-V 32-bit target ISA, processing binaries in ELF format and conducting an evaluation that includes static code size, code content, and code entropy analysis. The resulting assembly, presented in Intel syntax, is the only component suitable for persistence in a separate file. The objdump tool generates this assembly, which is then parsed and examined by the static analyzer.

The final evaluative layer pertains to execution investigations conducted with the instruction set simulator, ETISS [6]. Its flexibility in simulating any ISA at the instruction level makes it an indispensable tool. Binaries are executed within the simulator, with each instruction and its register values captured in a detailed runtime trace. This trace file is then further examined by the trace analyzer.

Results



Sizalizer uses the statistics and structures to graphically represent them for the target audience. The results are also used to suggest improvements. For Embench, for example, a replacement of 32-bit instructions with 16-bit compressed variants is proposed. In this case, the lw instruction ranks first among the 32-bit instructions found, both statically and dynamically. The immediate value is by far the most frequent value, with more than one million occurrences out of 2.5 million lw instructions executed, followed by the value 4. However, these values can also be represented in the smaller immediate range of a 16-bit compressed instruction. For such direct replacements, the improvement potential is also calculated for each of Embench's benchmarks, depicted in Figure **??**. Blue bars represent static and green bars dynamic improvement. The improvement potential is an upper bound on the code size reduction by completely replacing the given 32-bit instructions with their 16-bit variants. There is an improvement potential of 31 % for the static code size and 34 % for the dynamic code size.

References

[1] Christof Ebert and Capers Jones. "Embedded software: Facts, figures, and future". In: Computer 42.4 (2009), pp. 42–52.

[2] Charlotte Freitag et al. "The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations". In: Patterns 2.9 (2021).

[3] Andrew J Jarvis, David T Leedal, and C Nick Hewitt. "Climate–society feedbacks and the avoidance of dangerous climate change". In: Nature Climate Change 2.9 (2012), pp. 668–671.

[4] Ante Javor. Memgraph vs. Neo4j: A Performance Comparison. URL: `https://memgraph.com/blog/memgraph-vs-neo4j-performance-benchmark-comparison`. (accessed: 05.03.2024).

[5] Memgraph. URL: `https://memgraph.com/`. (accessed: 20.03.2024).

[6] Daniel Mueller-Gritschneder et al. "The Extendable Translating Instruction Set Simulator (ETISS) Interlinked with an MDA Framework for Fast RISC Prototyping". In: 2017 International Symposium on Rapid System Prototyping (RSP). 2017, pp. 79–84.