Profiling vs Static Analysis: The Impact on Precision Tuning

Profiling vs Static Analysis: The Impact on Precision Tuning

Lev Denisov
Politecnico di Milano
Milan, Italy
lev.denisov@polimi.it

Gabriele Magnani
Politecnico di Milano
Milan, Italy
gabriele.magnani@polimi.it

Daniele Cattaneo
Politecnico di Milano
Milan, Italy
daniele.cattaneo@polimi.it

Giovanni Agosta
Politecnico di Milano
Milan, Italy
giovanni.agosta@polimi.it

Stefano Cherubin
NTNU
Trondheim, Norway
stefano.cherubin@ntnu.no

ABSTRACT

In this study, we compare two different approaches to precision tuning — worst-case static annotation, and profile-guided annotation. We propose and implement a profile-guided approach in an existing precision tuning tool TAFFO. In comparison to the static approach used by the tool before, the profile-guided approach reaches up to 4 orders of magnitude lower error while providing comparable speedup and requiring less expertise from the programmer, tested on benchmarks of the PolyBench/C suite.

CCS CONCEPTS

• **Software and its engineering** → **Software design tradeoffs**; *Software performance*; **Compilers**; • **Computer systems organization** → *Embedded software*.

KEYWORDS

Precision tuning, compiler, performance evaluation, embedded systems, approximate computing

1 INTRODUCTION

Approximate computing is an umbrella term for the techniques that trade off accuracy for time-to-solution, energy-to-solution, or hardware complexity of compute-intensive applications. *Precision tuning* is an approximate computing technique notable for its wide applicability. This technique reduces the precision of the individual operations on real numbers in a given kernel, for instance from floating-point to fixed-point.

The conversion can be done manually, but it is an error-prone and tedious process that requires expertise and time. Alternatively, there exist tools that perform automated precision tuning. These tools typically use a *profile-guided* or *static* analysis to derive the code properties. In profile-guided analysis, the application code is instrumented and run on sample input data to collect information about the error propagation. The static analysis instead requires annotations and employs techniques such as dataflow analysis and symbolic execution to gather equivalent error propagation information.

While there are tools leveraging each approach [2], a comparison is hard to strike because different tools also entail differences in other steps of the precision tuning pipeline. It is an open question whether profile-guided or static analysis is more effective, and in which cases. The static analysis approach is intrinsically pessimistic, often producing results that are too conservative to be considered helpful for precision tuning. It also requires annotations, complicating the process. The quality of the profile-guided approach is dependent on the representativeness of the training data, but it also needs less effort from the programmer. In this paper, we address this question by comparing them within the same tool on the same set of benchmarks.

We chose as the starting point a state-of-the-art open-source precision tuning framework TAFFO [1], as it takes as input the intermediate representation of the industry-standard LLVM compiler framework (LLVM-IR), which ensures a degree of source language independence, as well as a wide applicability of the tool. We compare the two approaches on the full set of PolyBench/C benchmarks [3].

The contributions of this study: 1) we quantify the difference between static and profile-guided analysis for precision tuning in terms of error and speedup, 2) we provide a new profile-guided precision tuning tool that leverages the capabilities of the existing TAFFO framework.

2 PROFILE-GUIDED ANALYSIS

A crucial step of a precision-tuning process is the value range analysis, which characterizes the dynamic ranges of the program, and defines the possible level of their approximation. We propose a profile-guided value range analysis based on the instrumentation of the compiler intermediate representation code (LLVM-IR). The LLVM-IR file is first prepared by assigning unique names to all floating-point LLVM-IR registers and these names are saved into global constants. The logging function is also inserted into the LLVM-IR file. This function accepts the name of the register and its value. It performs either logging directly into a file or updates an in-memory structure that contains minimum and maximum values for the register name, and this structure is flushed into the log file later. After that the calls to the logging function are inserted for every floating-point register, passing the global constant with the register's name and the value of the register. The instrumented code is then compiled into a binary. The user can run the binary with one or more datasets, collecting the logged values in trace files, which are then used in the precision tuning process. The code changes done by the instrumentation are illustrated in listing 1 (original) and listing 2 (instrumented), the changes are highlighted in red.

Listing 1: Original LLVM-IR code

```
...
%var1 = fmul float %9, %12
%var2 = fadd float %10, %14
...
```

Listing 2: Instrumented LLVM-IR code

```
 @var1_name = constant "uniq1"
 @var2_name = constant "uniq2"
 define void @log_value(%name, %value) {...}
...
 %uniq1  = fmul float %9, %12
 call @log_value(@var1_name, %uniq1)
 %uniq2  = fadd float %10, %14
 call @log_value(@var2_name, %uniq2)
...
```

The process of mapping the dynamic ranges from the trace file back into the original program consists of a few steps. First, a dependency graph representing memory operations is built by analyzing the LLVM-IR of the program. Initially, the graph is seeded with one node for each instruction that allocates a buffer in memory (*alloca*, *malloc*, etc.). Then, all the uses of each buffer that do not change its value (*load*, *store*, usage as an argument to a function call, pointer operations, etc.) are added to the graph as new nodes, connected with an arc to the node that allocated the buffer. An example of such a cluster can be found in Listing 3.

Listing 3: A cluster of memory operations with the same range

```
%DY = alloca float
%"a::var123" = load float, float* %DY
store float %"a::var109", float* %DY
%DY2 = bitcast float* %DY to i8*
%"a::var109" = fdiv float 1.000000e+00, %"a::var108"
```

At the end of this process, each connected component (or *node cluster*) in the graph determines a set of virtual registers that shall share the same range. After building the node clusters, the trace files are parsed and the minimum and maximum values witnessed during execution are extracted for every register in the trace file. If a register belongs to a node cluster, these values are shared with all other registers in the cluster. At this point, the pass writes the just computed ranges into the LLVM-IR. If the user passes more than one trace file, the ranges are computed over all files combined. After that, any kind of data-type allocation and optimization strategies can be applied, regardless of the dynamic-range analysis employed before.

3 EXPERIMENTAL EVALUATION

In this section, we discuss the implementation of the profile-guided analysis in a state-of-the-art precision-tuning tool TAFFO and compare the profile-guided precision tuning with the static approach on the PolyBench/C benchmark suite.

3.1 Profile-guided TAFFO

Figure 1 illustrates the normal compilation flow and the compilation flows for TAFFO in static and profile-guided modes.

The normal compilation flow follows the standard set of transformations by the compiler front-end, middleware and back-end, from the source code, through LLVM-IR, ending in assembly code.

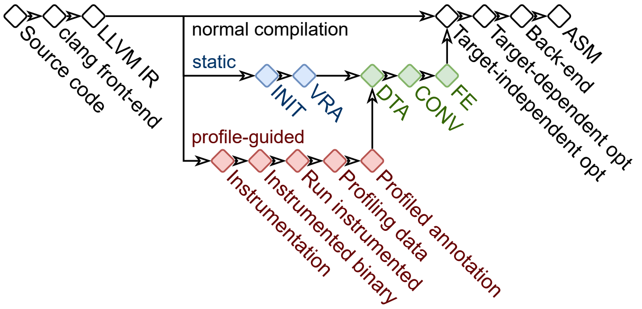

Figure 1: TAFFO compilation in static and profile-guided modes

The static mode in TAFFO is implemented as five LLVM analyses and transformation passes that happen right after the compiler front-end has produced the LLVM-IR (figure 1, static), namely: *Initializer* (INIT) reads the annotations and determines the amount of code affected by the tuning, *Value Range Analysis* (VRA) computes numerical intervals for all annotated variables and any other variables that depend on them, *Data Type Allocation* (DTA) decides the new data type for each intermediate value and variable, *Conversion* (CONV) modifies the LLVM-IR accordingly to the data type picked by DTA, *Feedback Estimator* (FE) statically estimates the error. For more details we refer the reader to Cattaneo et al. [1].

To implement profile-guided analysis we replace the static analysis passes of TAFFO (INIT and VRA) with the set of analysis steps implementing the methodology described in section 2 (figure 1, profile-guided). *Instrumentation* introduces the logging operations into LLVM-IR, which, in turn, is compiled into the *Instrumented binary*. *Run instrumented* step involves running the *Instrumented binary* with one or more datasets to collect *Profiling data*. *Profiled annotation* step applies the ranges from *Profiling data* into the original program. Then DTA, CONV, and FE stages of TAFFO are applied and the compilation resumes normally.

3.2 Experimental Setup

We evaluate Profile-guided TAFFO on the STM32L010 microcontroller with Arm Cortex-M0+ CPU (32 MHz), 128 Kbytes of Flash memory and 20 Kbytes of SRAM. Since this device lacks hardware support for floating-point, we use software implementation of floating-point operations with the *-msoft-float* compilation flag. The target is the PolyBench/C benchmark suite [3] version 4.2.1, featuring computational kernels from domains such as physics simulation, linear algebra, image processing, statistics, and dynamic programming. Due to the low amount of RAM available on the microcontroller, PolyBench was configured with the dataset size *MINI_DATASET*. We profile the benchmarks with the original datasets and then apply random noise to the dataset with amplitude up to ±100% of the original values for the evaluation.

3.3 Experimental Results

Figure 2 shows the mean relative error for the profile-guided and the static approaches. For both static and Profile-guided TAFFO the mean relative error is below 1% (10^{-2}) for the majority of the benchmarks. It can be also observed that Profile-guided TAFFO has lower or equal error than static in 24 out of 27 benchmarks. Moreover, in 13 out of 27 benchmarks the difference in error between static and

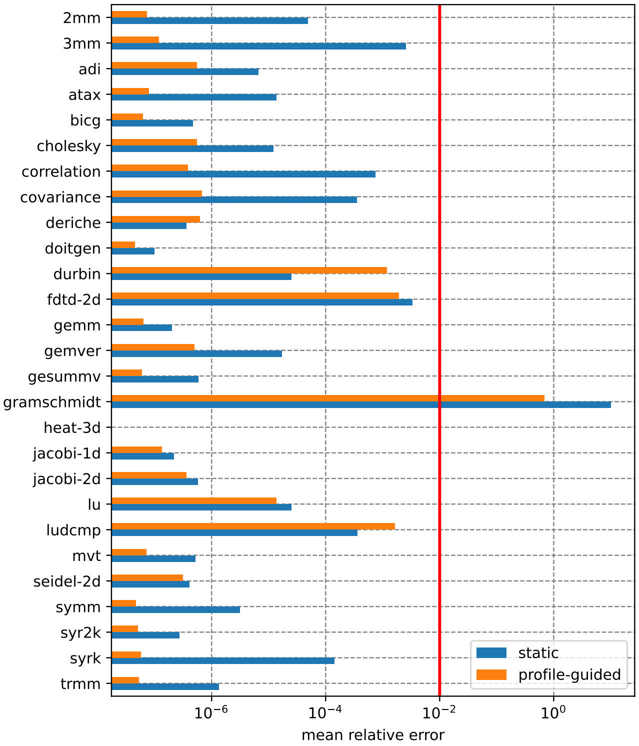

Figure 2: Mean relative error of static and Profile-guided TAFFO

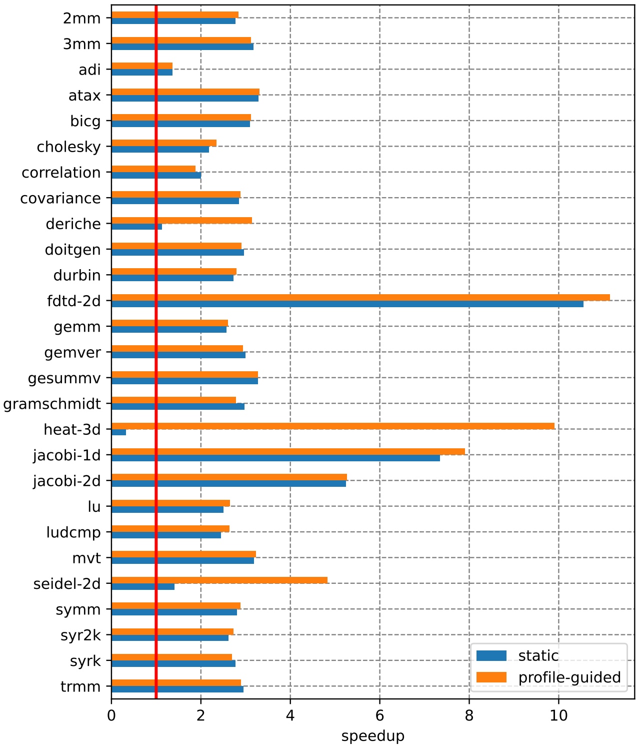

Figure 3: Speedup of static and Profile-guided TAFFO

Profile-guided TAFFO is an order of magnitude or higher. This can be explained by Profile-guided TAFFO achieving tighter ranges, leading to more bits being assigned to the fractional part, making it more accurate. In *heat-3d* the error is 0 for both approaches because this benchmark is very amenable to conversion into fixed-point arithmetic. Both static and Profile-guided TAFFO have a significant error in *gramschmidt* benchmark as it implements a numerically unstable algorithm that is not a good target for precision tuning optimization. Overall, Profile-guided TAFFO achieves much higher accuracy than static TAFFO: in more than 80% of benchmarks the error for Profile-guided TAFFO is less than 10^{-6}, while static TAFFO is below this threshold in only 40% of benchmarks.

Figure 3 shows the speedups relative to the unmodified float version. It can be seen that in 24 out of 27 benchmarks Profile-guided TAFFO achieves the same speedup as static TAFFO. In most cases, the speedup is in the range of 2-3×, with the occasional 5-10× speedup. There are 3 benchmarks where speedup obtained with Profile-guided TAFFO is significantly better than with static TAFFO: *deriche*, *heat-3d*, and *seidel-2d*. All of these cases can be explained by the static approach producing overly pessimistic ranges for the intermediate results, leading to significant parts of the program using conversions between the fixed-point and floating-point formats, leading to a smaller speedup or even a slowdown.

4 CONCLUSION

In this work, we explored the choice between static and profile-guided analysis in the construction of automated precision tuning tools. The experimental analysis shows that static and profile-guided analyses lead to similar performance impacts, but profile-guided analysis can significantly reduce the accuracy degradation.

ACKNOWLEDGMENTS

The authors gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, http://www.apropos.eu/).

REFERENCES

[1] Daniele Cattaneo et al. 2021. Architecture-aware Precision Tuning with Multiple Number Representation Systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 673–678. https://doi.org/10.1109/DAC18074.2021.9586303

[2] Stefano Cherubin and Giovanni Agosta. 2020. Tools for Reduced Precision Computation: a Survey. *Comput. Surveys* 53, 2 (Apr 2020), 35 pages. https://doi.org/10.1145/3381039

[3] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012), 1–1.