Automatic Differentiation of Compiled Programs

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of the Doctoral Degree
Doctor of Natural Sciences (Dr. rer. nat.)

to

Max Aehle

Date of Defense:	8th October 2024
Dean:	Prof. Dr. Christoph Garth
Reviewer:	Prof. Dr. Nicolas R. Gauger
Reviewer:	Prof. Dr. Christoph Garth
Reviewer:	Dr. Laurent Hascoët

DE-386

Abstract

Algorithmic differentiation (AD) is a set of techniques to "differentiate computer programs": Given a *primal program* that evaluates some mathematical function, AD allows to evaluate the derivative function. Unlike numerical difference quotients, algorithmic derivatives are accurate up to machine precision and, in the "reverse mode", can be formed with respect to a large number of input variables in one stroke. When combined with gradient-based optimization algorithms, AD is therefore a powerful tool to optimize engineering designs or learn weights of neural networks, besides many other applications. Up to date, most implementations of AD access the primal program via its *source code*, which they require to be available and written in a limited set of programming languages, typically not supporting the full language standards without manual user intervention.

In this dissertation, we present the novel AD tool *Derivgrind* that interacts with the *machine code* of the compiled primal program. Implemented in the Valgrind framework for dynamic binary instrumentation, Derivgrind augments portions of machine code with AD logic just in time before they potentially execute on the processor.

Specifically, Derivgrind's forward-mode AD logic keeps track of a floating-point "dot value" for every floating-point number appearing during the execution of the primal program. These dot values store the derivatives with respect to a single input variable, and are computed alongside, using elementary differentiation rules. Derivgrind also implements reverse-mode AD, by inserting AD logic that tags all floating-point numbers with identifiers, and uses them to record the real-arithmetic evaluation graph on a datastructure called the "tape".

Besides our extensive suite of regression tests and a simple numerical solver for Burgers' partial differential equation, we have tested Derivgrind on three larger software projects: the Python interpreter CPython, the spreadsheet software LibreOffice Calc, and the medical imaging application GATE based on the Monte-Carlo particle physics simulator Geant4. We are not aware of any successful previous attemps to compute algorithmic derivatives of these programs. With Derivgrind, only a few lines of code needed to be changed to accomplish that.

As a price for its versatility, Derivgrind slows down the primal program by a larger factor than many source-code-based tools. In addition, the issue of "bit-tricks" is more pronounced on the machine code level: If the primal program performs real-arithmetic operations in too obscure ways, Derivgrind cannot recognize their arithmetic meaning and may compute wrong derivatives. We have included a detailed discussion of various bit-trick mechanisms; in practical terms, nearly all of them are academic or originate from highly optimized math libraries. As long as differentiating those are avoided, Derivgrind is applicable to an unprecedentedly wide range of cross-language or partially closed-source software with little manual efforts.

Acknowledgements

My time as a PhD student in the past 3$^1/_2$ years has been exciting, filled with learning and discovering. I am deeply grateful to everyone who supported me and my scientific work presented in this thesis, and would like to use this page to express my thanks for the scientific support I received from so many people.

First and foremost, my sincere thanks go to my PhD supervisor Nicolas R. Gauger for his support and the freedom of research he provided. Being part of his interdisciplinary research group has enabled me, as a trained mathematician, to expand my knowledge into scientific computing and adjacent areas of computer science, and to do some meaningful work there. I want to thank Johannes Blühdorn and Max Sagebaum for our exchanges on algorithmic differentiation (AD), performance testing, and scientific software development; Ole Burghardt for introducing me to Valgrind-Memcheck from a user perspective and giving helpful advice on this thesis; and the entire group for the conversations that we have had in the past years.

Funding of my position by the State of Rhineland-Palatinate through the research training group SIVERT is gratefully acknowledged. Many thanks to all the PIs, and especially Ralf Keidel, for setting up and running this program, which balanced structure and guidance with room for creativity. And also to everyone else involved, especially my fellow PhD students! While some of our work on proton computed tomography did not make it into this thesis and the rest has been condensed into a single chapter, it had a lot of indirect impact on this thesis. For example, together we fixed some non-deterministic behaviour bugs related to uninitialized variables in an open-source project, and agreed that I prepare a talk about Valgrind for our internal seminar. This was the point when I looked under the hood and realized that the Valgrind framework can be used to differentiate machine code! And last but not least, SIVERT was the starting point for many further fruitful contacts and collaborations with interesting people.

The idea to try AD on the particle simulation framework Geant4 came up in meetings with Nico, Tommaso Dorigo, Pietro Vischia, and other members of the MODE collaboration. The size and complexity of the Geant4 codebase, and the resulting technical challenges when applying AD, were part of my motivation to build a machine-code-based AD tool. Mihály Novák has hosted my research stays at the CERN SFT group, and his excellent supervision helped us to understand, and partially overcome, the mathematical challenges of differentiating simulations like Geant4. Many thanks go to Laurent Hascoët, William Moses, and Vassil Vassilev, for our exchanges on AD and AD tools. Finally, thanks to everyone else who provided further input in meetings, at workshops and through peer reviews.

Contents

1. Introduction

1.1. Background

1.1.a. Algorithmic Differentiation. Algorithmic differentiation (AD) is a set of techniques to “differentiate computer programs”. When a computer program reads and writes real numbers, typically using digital floating-point formats, and computes the outputs from the inputs by a sequence of elementary arithmetic operations like $+$, $\cdot$, $\sqrt{\ }$, $|\cdot|$, and sin, it defines a function in the mathematical sense. As long as each of these elementary steps is differentiable, the chain rule of differential calculus allows to conclude that their composition is differentiable, and to calculate partial derivatives from elementary differentiation rules.[75,189]

1.1.b. Applications of AD. Though the key insight behind AD is thus simple, it has become a powerful tool indispensible for many applications. As a derivative with a large absolute value indicates that small perturbations of the input can lead to large changes in the output, they are useful for, e. g., *sensitivity analysis* and *uncertainty quantification*. As the gradient points into the direction of steepest ascend, many applications are related to *continuous optimization*, i. e., the identification of local minima or maxima of computer-implemented functions.

Better known as *backpropagation*, the *reverse mode of AD* has been implemented in machine learning (ML) frameworks like PyTorch[146] and TensorFlow[1], in order to evaluate gradients of the training error of ML models with respect to model parameters, such as weights in an artificial neural network; “learning” means minimizing the training error.

Various implementations of algorithmic derivatives in computational fluid dynamics codes[11,28,31,119,131,156,171,176] have served to improve aerodynamical objectives such as lift or drag, by proposing optimal airfoil shapes[11,33,35,122] or optimal flow actuation[134], besides many other industrial applications. It should be noted that in most of these studies, AD capabilities were integrated into an existing simulation code rather than rewriting it from scratch.

Many more studies have been reported in other application domains as diverse as ice sheet modelling[127], thermodynamics[76], molecular dynamics[186,187], inverse rendering[114,180], and quantitative finance[2,81].

Typically, AD-driven optimization studies allow for a logical split between

- the *primal program* implementing the objective function; it might have first been created without AD in mind;
- an *AD tool*, which has to interact with the primal program in order to identify the

elementary real-arithmetic steps it performs, in order to combine this information with elementary analytic differentiation rules to provide derivatives; and

- an *optimization framework*, which orchestrates the evaluation of the objective value and its derivatives at selected sets of inputs, and uses the results to steer the objective function towards a minimum or maximum.

1.1.c. Design Goals of AD Tools. There is a variety of AD tools in active use today, following different approaches on how to interact with the primal code and how to combine this information with the elementary differentiation rules.[1,27,29,79,91,123,128,158,163,179,184] Each AD tool finds its individual balance between conflicting goals like

- correctness of the computed derivatives,
- performance, usually in terms of increased run-time and memory consumption of the differentiated program compared to the primal program,
- scope of supported programs, in terms of requirements concerning access to the source code and build system of the program, supported programming languages, and supported parallel execution environments,
- support for advanced AD workflows like user-specified custom derivatives for certain subroutines,
- technical complexity of the AD tool, and
- the amount of experience and effort required from a user who wishes to apply AD to an existing computer program.

1.1.d. Most classical AD tools rely on source code. Most AD tools interact with the primal program through its source code, either by parsing it themselves (e. g. ADIFOR[27,29], TAPENADE[79]); by hooking into an existing compiler toolchain building the primal program (e. g. Clad[179], Enzyme[128]); as part of a runtime environment for a domain-specific language (e. g. Futhark[163]); or by means of programming language features like operator overloading (e. g. ADOL-C[184], CoDiPack[158], the autograd[123] tool used by PyTorch[146], or the internal tool of TensorFlow[1]), typically after some modification of the source code by the user. All of these approaches limit the scope of programs supported by the AD tool without major rewriting, making it difficult to apply AD to cross-language or partially closed-source software projects. To our knowledge, the AD tool Enzyme has the widest scope available so far: It can differentiate source code in any programming languages for which there is a front-end in the LLVM compiler structure (e. g. C, C++, Fortran, Rust, Julia). But, for instance, we are not aware of any AD tool immediately applicable to mixed Python-C++ programs.

To our knowledge, there has only been one published attempt to perform AD on the level of machine code so far: Gendler, Naumann and Christianson[69] have reported on the proof-of-concept prototype *adac*, which implements forward-mode AD by transforming the assembly code of the primal program.

1.2. This Dissertation

1.2.1. Research Goals

All programming languages eventually translate to machine code instructions executing on the processor. Thus, an AD tool operating on the machine code rather than the source code of the primal program can be expected to be applicable to a wider scope of programs, including cross-language and partially closed-source software.[69] Furthermore, with the reduced amount of required interaction with the source code of the primal program, applying a machine-code-based AD tool might be easier and quicker. In short, a robust and production-ready machine-code-based AD tool is likely to find a balance of the goals of Paragraph 1.1.c that is complementary to source-code-based AD tools, which might be preferable in some situations. The primary goal of this thesis is to provide the community with such a tool, along with a detailed analysis of its performance with respect to all of the goals of Paragraph 1.1.c for several large software projects.

Besides its application potential, machine-code-based AD is also interesting from a fundamental research perspective, as it faces a different set of challenges than source-code-based AD. Specifically, many real-arithmetic operations can be implemented in machine code by other means than the corresponding floating-point instructions, like a negative number being computed by a flip of the sign bit, without violating any language standard. Our second, sideline, research goal is therefore to get a practical overview of such *bit-tricks*, i. e. how they work, where they appear in relevant software projects, and how they can be detected.

1.2.2. Contributions

In this PhD project, we have created *Derivgrind*, a robust machine-code-based AD tool implementing both forward- and reverse-mode AD on the x86-64 Linux platform. Derivgrind employs the *dynamic binary instrumentation (DBI)* framework *Valgrind*[139,168] to insert differentiation logic into the machine code of the primal program while it executes. Naturally, in many cases, small portions of the source code of the primal program must still be accessible in order to identify the input and output variables. But as long as the intermediate real-arithmetic operations are performed by the same Linux process and comply with a few mild technical requirements related to the aforementioned bit-tricks, it does not matter from which and whose compilers or interpreters these instructions originate.

One of our validation examples demonstrates the versatility of machine-code-based AD in a very visual way: Derivgrind can differentiate the spreadsheet software LibreOffice Calc while it evaluates simple expressions entered by the user via the graphical user interface. The software repository of LibreOffice Calc contains a substantial amount of C++, Java and Python code, and the program was installed on the test system as a binary package by a standard software package manager. As Calc's add-on mechanism allows to access the relevant AD inputs and outputs from macro code supplied by the user at run-time, access to the source code of Calc was not required at all.

To demonstrate that Derivgrind is applicable to real-world scientific computing software, we provide derivatives in medical imaging and calorimetry setups based on the particle physics simulator Geant4[10,12,13] with only a few modified lines of code.

While performing these and a few other tests, we assembled a list of the mechanisms and occurrences of bit-tricks that we encountered. For the most important types of bit-tricks, we proposed and implemented ways to detect and handle them correctly. For the more obscure types of bit-tricks, however, this would either imply unreasonably high effort or be entirely impossible, as there are sometimes multiple meaningful ways to interpret a single sequence of machine code instructions as an (almost everywhere) differentiable real-arithmetic function. As a pragmatic alternative, we provide a "bit-trick finder" tool that can heuristically detect, and roughly localize, many types of bit-tricks.

1.2.3. Structure

1.2.3.a. Part I: Introduction. We use the first part of this thesis to introduce the "building blocks" of machine-code-based AD, starting with an overview on the key statements and methods concerning derivatives and algorithmic differentiation in Chapter 2. Chapter 3 is an introduction to the x86-64 instruction set architecture with an emphasis on the floating-point representation of real numbers, arithmetic operations, and elementary functions. In Chapter 4, we give an overview on DBI frameworks and tools and take a detailed look at the Valgrind framework.

1.2.3.b. Part II: Assembling the Novel AD Tool Derivgrind. In the second part of this thesis, we use these "building blocks" to assemble the novel AD tool Derivgrind. We start with the forward mode in Chapter 5, focusing on the technical aspects because the forward-mode AD logic in itself is simple. In Chapter 6, we enable reverse-mode AD by adding tape-recording capabilities to our tool. Chapter 7 deals with a heuristic to detect and approximately localize many types of bit-tricks in the primal program.

While the forward-mode, tape-recording, and bit-trick-finding logic differ in many details, their overall structure is similar, as all three of them utilize a particular type of shadow data, process it in a similar way, and define so-called monitor commands, client requests and math wrappers. Readers are invited to go through Chapters 5 and 6 in parallel.

1.2.3.c. Part III: Evaluation of Derivgrind. We have extensively tested Derivgrind with respect to the design goals listed in Paragraph 1.1.c. The validation Chapter 8 introduces our regression test suite, which checks the correctness of the computed derivatives for many small sample client programs. Additionally, two applications to complex client programs performing simple arithmetic operations, CPython and LibreOffice Calc, are presented. In the performance Chapter 9, we measure the run-time and memory complexity of Derivgrind, and compare it with the AD tool CoDiPack, using a benchmark based on Burgers' partial differential equation. In Chapter 10, we give more context on the interdisciplinary project in which Derivgrind was developed.

We close the third part with an application of Derivgrind to complex software performing complex arithmetic: In Section 10.3, as a technical feasibility study, we compute derivatives in a medical imaging setup provided by the Bergen pCT collaboration[14] using the program GATE[100] based on the particle physics simulator Geant4[10,12,13]. Finally, in Section 10.4, we analyse the mathematical challenges of applying AD to this kind of stochastic codes for a simple calorimetry setup, first using the more compact simulation toolkit G4HepEm[142] and then going back to Geant4 again.

1.2.3.d. Conclusion and Appendices. Chapter 11 summarizes this work and discusses possible future research directions. Appendix A gives more background on the rare setups in this thesis where Derivgrind did not provide correct derivatives due to unsupported bit-tricks. Some relevant but long code snippets have been collected in Appendix B.

1.2.4. Code

The source code of the Derivgrind package is available at

`https://github.com/SciCompKL/derivgrind`.

Most of our changes and additions to the Valgrind framework have been released under the GNU General Public License version 2 or later. A few small portions of the package, including client request headers and wrappers that might end up being included in a client program, have been released under permissive licenses.

Furthermore, the code of the shadow memory tool used by Derivgrind is available at

`https://github.com/SciCompKL/flexible-shadow`

under the MIT license.

1.2.5. Prior Publications

We have previously published major results of this work in preprints or journal publications; some parts with major similarities in logical structure or wording, when we considered those optimal. The following list gives an overview about the relevant publications and preprints, leaving out talks.

- We have published the idea of leveraging dynamic binary instrumentation and Valgrind to implement forward-mode AD in the paper *Forward-Mode Automatic Differentiation of Compiled Programs*[4] together with Johannes Blühdorn, Max Sagebaum, and Nicolas R. Gauger. This paper gives an introduction to the relevant prerequisites (Section 2.3.1, Paragraph 3.1.a), describes the instrumentation, user interface and math wrappers (Chapter 5), and reports on our regression and performance tests (Sections 8.1, 8.2 and 9.2). Figures 5.5, 9.1 and 9.2 have been cited from the paper.

- In the paper *Reverse-Mode Automatic Differentiation of Compiled Programs*[5] by the same authors, we add reverse-mode capabilities (Section 2.4.1) to the Derivgrind package (Chapter 6), and measure its performance (Section 9.3). Figures 6.3 and 9.3 to 9.5 have been adapted from this work.
- The preprint *Derivatives in Proton CT*[3] and the subsequent journal article *Exploration of Differentiability in a Proton Computed Tomography Simulation Framework*[6] assess the potential of AD for optimization and uncertainty quantification in a medical imaging setup; this work has been adapted into Sections 10.1.3 and 10.2, and Figures 2.1, 2.2 and 10.2 to 10.4 have been cited or adapted from it.
- We have contributed the section *Automatic Differentiation of GATE/Geant4* to the paper *Progress in End-to-End Optimization of Particle Physics Instruments with Differentiable Programming*[7], where we apply Derivgrind to the Monte-Carlo particle simulation toolkit Geant4 in order to demonstrate that Derivgrind can solve this technical challenge. We repeat the methodology and results in Section 10.3, with Figures 10.5 and 10.6 cited from the paper.
- In the paper *Optimization Using Pathwise Algorithmic Derivatives of Electromagnetic Shower Simulations*[8] with Mihály Novák, Vassil Vassilev, Nicolas R. Gauger, Lukas Heinrich, Michael Kagan, and David Lange, we have assessed mathematical challenges related to applying AD to a particle simulation in high-energy physics; our results have been summarized in Section 10.4, using Figures 10.8 to 10.12 from the paper.

2. Algorithmic Differentiation

2.1. Real Analysis Recap: Derivatives

2.1.a. Definition. Let $U \subset \mathbb{R}^n$ be an *open* subset of $\mathbb{R}^n$; i. e., for every element $x^* \in U$ there is a $\delta > 0$ such that the entire ball $\{x \in \mathbb{R}^n : \|x - x^*\| < \delta\}$ of radius δ around x^* is contained in U.

Let $f : U \to \mathbb{R}^m$ be a (mathematical) function, $x^* \in U$, and $L : \mathbb{R}^n \to \mathbb{R}^m$ be a linear map. We say that f *is differentiable at x^* with derivative L* if the *affine-linear approximation error*

$$\begin{aligned} r : U &\to \mathbb{R}^m \\ x &\mapsto f(x) - f(x^*) - L(x - x^*) \end{aligned} \tag{2.1}$$

becomes so small around x^* that

$$\frac{\|r(x)\|}{\|x - x^*\|} \to 0 \qquad \text{for } x \to x^*,\ x \neq x^*. \tag{2.2}$$

The choice of the norm $\|\cdot\|$ is irrelevant here, as for any pair of norms on $\mathbb{R}^n$, one is bounded by a constant multiple of the other. Differentiability of f at x^* implies continuity of f at x^*, i. e., $f(x) \to f(x^*)$ for $x \to x^*$.

If derivatives exist, they are unique. To prove this, suppose that f is differentiable at x^* with derivatives L_1 and L_2, and that $L_1 v, L_2 v \in \mathbb{R}^m$ differ in their j-th component for some $v \in \mathbb{R}^n$. Obviously, $v \neq 0$, as the contrary would imply $L_1 v = 0 = L_2 v$. By (2.2), the j-th components of

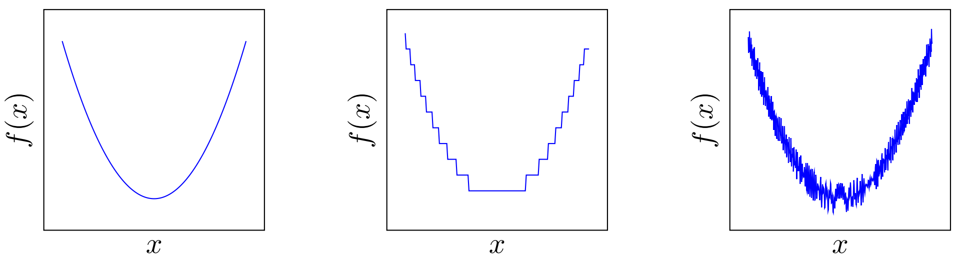
$$\frac{f(x) - f(x^*) - L_1(x - x^*)}{\|x - x^*\|} \qquad \text{and} \qquad \frac{f(x) - f(x^*) - L_2(x - x^*)}{\|x - x^*\|}$$

go to 0 for $x \to x^*$, $x \neq x^*$. Forming the difference and choosing $x = x^* + \alpha v$, which is contained in U for any sufficiently small $\alpha > 0$ because U is open, we obtain that

$$\frac{-L_1(\alpha v)_j + L_2(\alpha v)_j}{\|\alpha v\|} \to 0 \tag{2.3}$$

for $\alpha \to 0$, $\alpha > 0$. As the left side, expressed as $\frac{-(L_1 v)_j + (L_2 v)_j}{\|v\|}$, is independent of α and not zero, this leads to a contradiction. □

Thus, L is unique and we may call it "the" derivative of f at x^*, and denote it as $f'(x^*)$. The representation of this linear map as a matrix in $\mathbb{R}^{m \times n}$ is called *Jacobian matrix.* In the following, we do not distinguish between linear maps and their matrix representations with respect to standard bases.



(a) Good case. (b) Function with jumps. (c) Function with noise.

Figure 2.1: Gradient descent finds the global minimizer for the smooth convex function in (a), if a suitable step-size is chosen. Jumps as in (b), or noise as in (c), may however cause trouble. Adapted from Aehle et al.[6]

2.1.b. Usage. By the above definition of differentiability, knowing $f'(x^*)$ gives us a sense of "how f looks like around x^*": Namely, it can be approximated by its *first-order Taylor expansion*, which is the affine-linear function

$$\begin{aligned} \mathbb{R}^n &\to \mathbb{R}^m \\ x &\mapsto f(x^*) + f'(x^*) \cdot (x - x^*). \end{aligned} \tag{2.4}$$

This fact is exploited by gradient-based numerical optimization algorithms like the *gradient descent* algorithm, which attempts to find a minimizer of a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$ by iteratively improving a "candidate minimizer" x^*, adding a multiple of $-f'(x^*)^T$ in each step because that is the direction in which (2.4) decreases most steeply. Gradient descent converges to a local and/or global minimizer under sufficiently strong conditions on f; it should be clear that jumps or noise, as shown in Figure 2.1, may diminish the value of derivative information even if f is differentiable almost everywhere.

Gradient-based optimization algorithms form the "back-end" of many engineering design optimization, parameter fitting, and machine learning applications. We consider them to be the main reason why tools to differentiate computer programs are important.

2.1.c. Computation. Derivatives of many elementary mathematical operations and functions are well-known as *differentiation rules.* For example,

- for all $\lambda_1, \lambda_2 \in \mathbb{R}$, the *linearity of differentiation* tells us that the derivative of

$$\begin{aligned} f : \mathbb{R}^2 &\to \mathbb{R} \\ \tbinom{x_1}{x_2} &\mapsto (\lambda_1 x_1 + \lambda_2 x_2) \end{aligned} \tag{2.5}$$

  is $f'\tbinom{x_1^*}{x_2^*} = (\lambda_1, \lambda_2)$,

- the *product rule* states that the derivative of

$$\begin{aligned} f : \mathbb{R}^2 &\to \mathbb{R} \\ \tbinom{x_1}{x_2} &\mapsto (x_1 \cdot x_2) \end{aligned} \tag{2.6}$$

is $f'\binom{x_1^*}{x_2^*} = (x_2^*, x_1^*)$, and

- the derivative of the natural logarithm

$$f = \ln : \mathbb{R}_{>0} \to \mathbb{R}$$

is $f'(x^*) = \frac{1}{x^*}$.

For mathematical expressions composed from elementary operations and functions, the derivative can be formed using the *chain rule*: For open sets $U \subset \mathbb{R}^n$ and $V \subset \mathbb{R}^m$, and functions $f : U \to V$ and $g : V \to \mathbb{R}^\ell$ differentiable in $x^* \in U$ and $f(x^*) \in V$, respectively, the composition

$$\begin{aligned} g \circ f : U &\to \mathbb{R}^\ell \\ x &\mapsto g(f(x)) \end{aligned}$$

is differentiable in x^* with $(g \circ f)'(x^*) = g'(f(x^*)) \cdot f'(x^*)$.
To prove this, we have to show that

$$\frac{\|g(f(x)) - g(f(x^*)) - g'(f(x^*)) \cdot f'(x^*) \cdot (x - x^*)\|}{\|x - x^*\|} \to 0 \quad \text{for } x \to x^*,\ x \neq x^*. \tag{2.7}$$

Denoting the affine-linear approximation error (2.1) of f at x^* by r, we may rewrite $f'(x^*) \cdot (x - x^*)$ as $(f(x) - f(x^*) - r(x))$ in the numerator. The term $r(x)$ may be left out, because

$$\frac{\|g'(f(x^*)) \cdot r(x)\|}{\|x - x^*\|} \leq \|g'(f(x^*))\| \cdot \frac{\|r(x)\|}{\|x - x^*\|} \to 0, \tag{2.8}$$

where $\|g'(f(x^*))\|$ denotes the operator norm (maximal scaling factor) of $g'(f(x^*))$, which is a finite constant. So instead of (2.7), it suffices to show that

$$\frac{\|g(f(x)) - g(f(x^*)) - g'(f(x^*)) \cdot (f(x) - f(x^*))\|}{\|x - x^*\|} \to 0 \quad \text{for } x \to x^*,\ x \neq x^*. \tag{2.9}$$

As this is trivially satisfied for those sequence elements x that satisfy $f(x) = f(x^*)$, we may in the following assume that $f(x) \neq f(x^*)$. The differentiability of f additionally ensures that

$$\|f(x) - f(x^*)\| = \|f'(x^*) \cdot (x - x^*) + r(x)\| \leq \left(\|f'(x^*)\| + \frac{\|r(x)\|}{\|x - x^*\|}\right) \cdot \|x - x^*\|. \tag{2.10}$$

As the first factor on the right hand side is bounded, we may replace the denominator $\|x - x^*\|$ in (2.9) by $\|f(x) - f(x^*)\|$ and obtain

$$\frac{\|g(f(x)) - g(f(x^*)) - g'(f(x^*)) \cdot (f(x) - f(x^*))\|}{\|f(x) - f(x^*)\|} \to 0 \quad \text{for } x \to x^*,\ f(x) \neq f(x^*) \tag{2.11}$$

as a sufficient condition. This statement follows from the differentiability of g and the continuity of f, completing the proof. □

2.1.d. Numerical Approximation. In the case of $n = m = 1$, i. e. $f : \mathbb{R} \to \mathbb{R}$, (2.2) simplifies to the statement that the derivative $L = f'(x^*)$ is the limit of the difference quotient,

$$\frac{f(x) - f(x^*)}{x - x^*} \to L \quad \text{for } x \to x^*,\ x \neq x^*. \tag{2.12}$$

Numerically, the value of the difference quotient for some value $x = x^* + h$ or $x = x^* - h$, with a small $h > 0$, can be used as an approximation of the derivative. For twice continuously differentiable functions, the numerical error of this approximation, called *truncation error*, is at most proportional to h for $h \to 0$. For three-times continuously differentiable functions, the truncation error of the *central difference quotient*

$$\frac{f(x^* + h) - f(x^* - h)}{2 \cdot h} \tag{2.13}$$

is at most proportional to h^2 for $h \to 0$.

While smaller choices of h reduce the truncation error, they increase the condition number of the difference quotients. Often, the function f can only be evaluated with a small error or uncertainty, e. g. because f includes making physical measurements or computations in floating-point arithmetic. In this case, h cannot be chosen arbitrarily small, because the two terms in the numerator of (2.13) have about the same magnitude, so taking their difference reduces the number of significant digits.

If $m > 1$ but $n = 1$, the difference quotients in (2.12) and (2.13), now composed of vector operations, approximate the derivative $L \in \mathbb{R}^{m \times 1}$ with only two evaluations of f. If $n > 1$, the difference quotients must be formed with respect to n separate directions. Thus, the number of operations needed to approximate L scales proportionally with the number n of inputs.

2.2. Computer Programs as Mathematical Functions

According to the chain rule, it is possible to assemble the derivative of an arbitrarily complicated mathematical function from the derivatives of the involved elementary operations. This task can be carried out by hand, with the help of a computer algebra system, or by the algorithmic differentiation (AD) techniques that we will explain in the following sections. First, however, let us describe the way in which the mathematical function is represented in the context of AD.

2.2.a. Primal Program. AD evaluates derivatives of mathematical functions implemented by computer code, which we refer to as the *primal program*. At this point, the term “program” should be understood in a wide sense, and may refer to interpreted scripts, source code of executables or software libraries, or the respective binary build artifacts.

2.2.b. Input and Output. Primal programs can read and write digital approximations of real numbers, encoded e. g. as decimal strings or in a binary floating-point format, Listing 2.1: A simple differentiable function, given by its source code in the programming language C.

```
#include <stdio.h>
#include <math.h>

double f(double x1, double x2):
  printf("some side effect")
  double tmp = x1 * 3.14 + sin(x2);
  if(x2>x1){
    return tmp * x2;
  } else {
    return x2 - 1.0;
  }
}
```

through function call interfaces, standard and file input/output streams, and other mechanisms. Unlike "pure" mathematical functions, the execution of a primal program may have side effects such as modifying an internal state or cache file used by the primal program.

2.2.c. Programs as Sequences of Real-Arithmetic Steps. No matter how complex the functionality of a computer programs is, it computes real-number outputs from real-number inputs by a sequence of simple, elementary, real-arithmetic processing steps

$$a_{\text{lhs}} = \phi(a_1, \ldots, a_k), \tag{2.14}$$

such as copy operations, basic real-arithmetic operations $(+, -, \cdot, /)$, and math functions $(|\cdot|, \sqrt{\ }, \sin, \exp, \ldots)$. Assuming at this point that it is possible to extract this sequence of real-arithmetic steps, we have turned the primal program into a mathematical function f.

The set of real-arithmetic functions eligible to appear as a ϕ on the right-hand side of (2.14) can vary, but typically they are all differentiable "almost everywhere" across their respective domains. Heuristically, we will "almost surely" not hit operand values where elementary operations are not differentiable, and can thus conclude by the chain rule that the composition $f : \mathbb{R}^n \to \mathbb{R}^m$ is differentiable for "most" inputs to the primal program. See Paragraph 2.2.e below for a hint to a more formal perspective.

For example, the C code in Listing 2.1 computes the return value by means of a scaling operation and an evaluation of the sine function, followed by an addition, eventually followed by a multiplication if the second argument is larger than the first argument. Otherwise, a different sequence of operations applies. The sequence of performed real-arithmetic steps (2.14) may depend on the input value x^* where the derivative is to be evaluated.

Integer arithmetic, comparisons, boolean logic, and control flow constructs like function calls, conditional branches and loops are ignored here. While those may affect *which*

Listing 2.2: This function behaves like an identity function on floating-point inputs. However, it is nearly impossible for an AD tool to recognize this.

```
#include <stdio.h>

double f(double x):
  double y;
  char str[100];
  sprintf(str, "%.80e", x);
  sscanf(str, "%lf", &y);
  return y;
}
```

sequence of real-arithmetic calculations is performed, for any particular set of inputs x^*, it is only these real-arithmetic calculations that determine the AD derivative at x^*.

Note that our assumption made above, that any processing of floating-point data, from the real inputs to the real outputs, can be divided into elementary steps with obvious real-arithmetic meaning, is not always guaranteed. For instance, the code in Listing 2.2 converts the input argument into a decimal string, and parses this string into a floating-point number to obtain the return value; automatically recognizing this as a real-arithmetic identity would be very hard. Later in this thesis (Section 3.3.5 and Paragraph 7.5.a), we will even encounter an example for computer code that has two different interpretations as a real-arithmetic step, with different derivatives; let us postpone the proper analysis of such “bit-tricks” to Section 3.3.

2.2.d. Digression: Derivatives of Numerical Algorithms. While the pure mathematical theory of a real-world process may be smooth, numerical software often only computes approximations. Even if the error of the computed value is small, the error of the derivatives may be large.[70] The example in Figure 2.2, cited from Aehle et al.[3], demonstrates this for an iterative solver of a polynomial equation of degree 3: Starting from any initial value in the appropriate range, the solver computes a good approximation of the real root; but as the approximation is noisy, its derivative with respect to the initial value is sometimes much larger than 0.

When a subprocedure of f is known to solve a mathematically simple numerical problem, it often makes sense to use analytic derivatives for this subprocedure instead of calculating a “black-box” derivative of the numerical algorithm. This has been worked out, e. g., for the solution of a linear system,[60] the non-degenerate dominating eigenvalue and eigenvector of a diagonalizable matrix,[195] and the limit of a fixed-point iteration (where this is known as *reverse accumulation*).[42] There is a nice overview article about AD “pitfalls” by Hückelheim et al.[85]

Of course, implementation details and possible benefits depend on the specific numerical algorithm, and on its context in the primal program. For the scope of this thesis, we will not use any higher-level information on the algorithm composed by the sequence of elementary real-arithmetic operations (2.14).

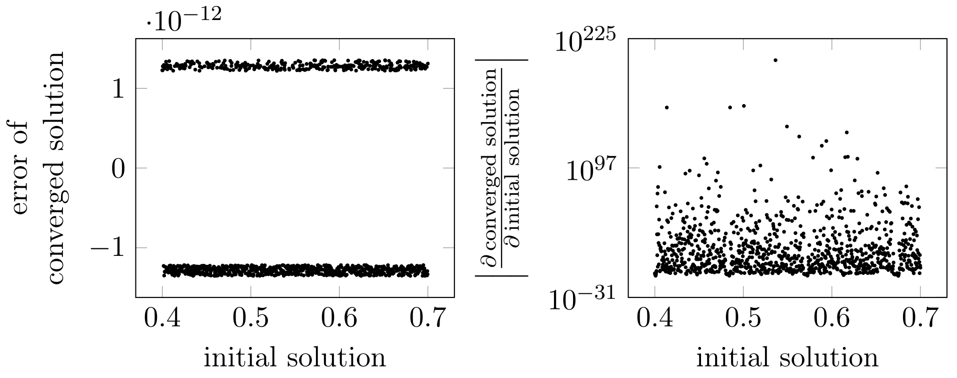


Figure 2.2: While the damped Newton scheme $x^{(k+1)} = x^{(k)} - \alpha g(x^{(k)})/g'(x^{(k)})$ converges to the single real solution of $g(x) = x^3 - 2x + 2$, the derivative of the converged solution with respect to the initial solution $x^{(0)}$ is far from 0. Here, $\alpha = 0.1$ and the solver was stopped as soon as $|g(x^{(k)})| < 10^{-11}$. Cited from Aehle et al.[3]

2.2.e. Digression: PAP Functions. The heuristic argument that if elementary operations ϕ_1, ϕ_2 are differentiable almost everywhere (in the sense that there is only a measure-zero set of counterexamples), then the composition $\phi_2 \circ \phi_1$ is differentiable almost everywhere, has an obvious flaw: ϕ_1 may take a constant value, at which ϕ_2 is not differentiable, on a set with positive measure. To formalize the key insight that occasional jumps or kinks of some elementary functions do not prevent us from assigning meaningful derivatives, Lee et al.[111] have introduced the concept of functions that are *piecewise analytic under analytic partition* (PAP). Many elementary functions are PAP functions, and compositions of PAP functions are PAP functions. Every PAP function has a set of *intensional derivatives*, which are PAP functions again and agree with the standard derivatives almost everywhere.

2.2.f. Next Sections. Implementations of *AD tools* need to answer two crucial questions:

- Which types of primal programs (Paragraph 2.2.a) can be handled by the AD tool, and how does the AD tool find out about the sequence of real-arithmetic steps like (2.14)? In Sections 2.5 to 2.7, we give an overview on the different mechanisms reported in the literature to access the primal program.
- Once we know which real-arithmetic steps a program performs, how can we combine this information with the elementary differentiation rules and the chain rule to compute derivatives? We summarize the forward and reverse modes of AD in Sections 2.3 and 2.4.

As part of our general overview, we will have a few specific looks at the AD tools Tapenade[79], CoDiPack[158], Enzyme[128] and Clad[179]. These tools, and many others, are well-established, excellent choices, and power numerous fruitful applications of AD. It is in the nature of things that in a thesis concerned with a novel approach to a problem, special attention is given to weaknesses in the existing successful approaches. Whenever we do so, our purpose is to work out some specific problems of source-code-based AD that machine-code-based AD can solve, and to contextualize the results of our equally critical and “picky” assessment of our novel machine-code-based AD tool Derivgrind.

2.3. Forward-Mode AD: Propagation of Dot Values

Let us assume that we already know the real-arithmetic steps (2.14) performed by the primal program; mechanisms to obtain this information will be discussed in Sections 2.5 to 2.7. The *forward* and *reverse mode* of AD are two ways of using the chain rule to assemble the elementary derivatives into the derivative of the primal program. This section is about the (easier) forward mode, and the next Section 2.4 is about the reverse mode.

2.3.1. Basic Procedure

Let us first consider the case of a single AD input x. For every number a appearing in one of the real-arithmetic steps of the primal program, the forward mode of AD keeps track of a *dot* or *tangent* value $\dot{a}$ that stores its partial derivative $\frac{\partial a}{\partial x}$ with respect to the single input x. Initially, $\dot{x}$ is *seeded* with a value of 1, and all other dot values (e. g. those of global variables or constants) are initialized with 0. Then, every real-arithmetic step like (2.14) is matched by an update

$$\dot{a}_{\text{lhs}} = \frac{\partial \phi}{\partial a_1}(a_1, \ldots, a_k) \cdot \dot{a}_1 + \cdots + \frac{\partial \phi}{\partial a_k}(a_1, \ldots, a_k) \cdot \dot{a}_k \tag{2.15}$$

according to the chain rule. This update uses the differentiation rule for the operation ϕ to compute $\dot{a}_{\text{lhs}}$ from $\dot{a}_1$, …, $\dot{a}_k$. For instance, a multiplication $a_{\text{lhs}} = a_1 \cdot a_2$ leads to an update $\dot{a}_{\text{lhs}} = \dot{a}_1 \cdot a_2 + a_1 \cdot \dot{a}_2$ of the dot value of the product, and $a_{\text{lhs}} = \sin(a_1)$ has to be accompanied by $\dot{a}_{\text{lhs}} = \cos(a_1) \cdot \dot{a}_1$.

After the last real-arithmetic step and the associated forward-mode AD logic (2.15) have been performed, the derivatives of all of the AD outputs with respect to the single AD input x can be read from their respective dot values.

2.3.2. Multiple Inputs

In advanced use cases with multiple AD inputs x_1, …, x_n, their dot values can be seeded with $\dot{x}_1 = \nu_1$, …, $\dot{x}_n = \nu_n$ in the beginning. As both differentiation and the updates according to (2.15) are linear, the dot value $\dot{a}$ of a value a then represents

$$\dot{a} = \nu_1 \cdot \frac{\partial a}{\partial x_1} + \cdots + \nu_n \cdot \frac{\partial a}{\partial x_n}. \tag{2.16}$$

This fact allows to compute a single directional derivative with the same computational effort as a partial derivative with respect to a single AD input. This is sometimes called *jacobian-vector product (JVP)*.

If partial derivatives with respect to multiple AD inputs are sought, these can be computed separately, each of them as in the case with a single AD input. Alternatively, in *vector-mode* forward-mode AD, $\dot{a}$ is a vector containing the partial derivatives with respect to all AD inputs, and (2.15) is applied in each component. Both approaches can be combined. Either way, the computational effort is proportional to the number of AD inputs.

2.3.3. Asymptotical Performance

Already at this abstract stage, we can make asymptotic estimations about how the run-time and memory performance of the differentiated program relates to the performance of the primal program. Forward-mode AD with a single AD input needs to store one additional real value $\dot{a}$ for every real value a appearing in the primal program. If a large share of the program's memory is devoted to floating-point data, the memory consumption can therefore be expected to scale by a factor of 2.

Regarding the run-time, one additional real-arithmetic step (2.15) needs to be performed for every real-arithmetic step (2.14) in the primal program, and for every AD input. Therefore, the run-time scales at most by the number of inputs times a factor independent of the primal program. This factor depends on details of the AD tool implementation and the computing system, such as

- how fast (2.15) can be evaluated compared to (2.14), for all of the possible elementary real-arithmetic steps (which form a finite set);
- how much time is needed by the AD tool to identify the real-arithmetic statements like (2.14), and to form (2.15) — often, this happens statically and does not incur any slow-down at run-time; and
- how much the inserted logic (2.14) obstructs performance optimizations of the compiler (e. g., use of vector registers) and the processor (e. g., pipelining and caching).

While the forward-mode run-time scales proportionally with the number of AD inputs, it does not explicitly depend on the number of AD outputs; this is the same asymptotic run-time behaviour as with difference quotients (Paragraph 2.1.d). In the context of gradient-based optimization (Paragraph 2.1.b) with a single AD output given by the objective value, and a possibly large number of parameters treated as AD inputs, the other way round would be much better. Fortunately, there is a *reverse mode* of AD that we will take a look at next.

2.4. Reverse-Mode AD: Backpropagation of Bar Values

2.4.1. Basic Procedure

Let us first consider the case of a single AD output y. In this case, reverse-mode AD matches every real-arithmetic value a appearing in the evaluation of the primal program by an *adjoint* or *bar variable* $\bar{a}$. In $\bar{a}$, the partial derivative $\frac{\partial y}{\partial a}$ of y with respect to a

is accumulated. Initially, $\bar{y}$ is seeded with a value of 1, and all other bar values (e. g. constants, global variables, AD inputs, and intermediate results) are initialized with 0. Then, all real-arithmetic steps like (2.14) are taken into account *in the reverse order of execution* compared to the primal program, updating

$$\bar{a}_i \mathrel{+}= \frac{\partial \phi}{\partial a_i}(a_1, \ldots, a_k) \cdot \bar{a}_{\text{lhs}} \qquad \text{for } i = 1, \ldots, k, \text{ and} \tag{2.17}$$

$$\bar{a}_{\text{lhs}} = 0. \tag{2.18}$$

The increment in (2.17) reflects that the derivative of the output variable y with respect to the value of a_i *before* the step differs from the derivative with respect to the value of a_i *after* the step by the additional implicit dependency of y on a_i via a_{lhs}. Equation (2.18) refers to the fact that the overwritten value of a_{lhs} *before* the assignment has no influence on the output variable y.

After the update (2.17), (2.18) has been performed for all real-arithmetic steps, the derivatives of the single AD output y with respect to all of the AD inputs can be read from their respective bar values.

2.4.2. Multiple Outputs

The case of multiple AD outputs in the reverse pass is analogous to the case of multiple AD inputs in forward-mode AD (Section 2.3.2), with swapped roles of AD inputs and outputs.

In advanced use cases for multiple output variables y_1, …, y_m, their bar values can be seeded with $\bar{y}_1 = \mu_1$, …, $\bar{y}_m = \mu_m$. Then, the bar value $\bar{a}$ of a value a accumulates

$$\bar{a} = \mu_1 \cdot \frac{\partial y_1}{\partial a} + \cdots + \mu_m \cdot \frac{\partial y_m}{\partial a}, \tag{2.19}$$

allowing to compute a single gradient of a linear combination of output values with the same computational effort as for a single output value. This is sometimes called *vector-jacobian product (VJP).*

If separate gradients of multiple AD outputs are sought, they can be computed by individual reverse passes, by a single vector-mode reverse pass, or a combination of both approaches. In either case, the computational effort scales with the number of AD outputs.

2.4.3. Forward and Reverse Pass

In forward-mode AD, the dot value propagation logic (2.15) can be performed alongside the original statements like (2.14). In contrast, the reverse-mode updates (2.17) and (2.18) have to be applied to the statements in reverse order, compared to the order in which they are performed by the primal program. Reversing the control flow is not trivial:

- At the beginning of the reverse-mode AD computation, the partial derivatives of the last statement are needed to perform the update (2.17). In general, they depend on the final state of the primal program, so the primal program must be run to the end beforehand.

- Next, the partial derivatives of the second-to-last statement are needed for (2.17). In general, they depend on the second-to-final state of the primal program, which is not available any more once the primal program has been run to the end. A naive approach to obtain the second-to-final state could thus be to run the primal program another time, until the second-to-last statement.
- But then, the same problem occurs with the third-to-last statement, and so on.

Asymptotically, such a *recompute-all* approach squares the number of real-arithmetic statements to be executed, which is clearly not efficient. Reverse-mode AD is usually implemented with a run-time proportional to the run-time of the primal program, by running a *forward pass* before the *reverse pass* procedure of Section 2.4.1. In the forward pass, the primal program is run with additional AD logic that records information required to make the reverse pass efficient. This forward-pass data structure is often called *tape*, *stack* or *cache*, and is mainly accessed in a last-in-first-out fashion. We discuss two common approaches to the forward pass in Sections 2.4.4 and 2.4.5.

2.4.4. Source-Rewriting Approach

Listing 2.3 gives an example of how the AD tool Tapenade[79] performs the forward and reverse passes. The primal program is defined by the function `f`, which repeatedly multiplies the argument `u` by the argument `v` until it becomes larger than 100, and then returns `u`. Tapenade generates source code of a function `f_b` that takes three additional parameters `ub`, `vb` and `fb`. The caller of this function has to pass the bar value of the return value of `f` in `fb`, and pointers to the bar values of `u` and `v` in `ub` and `vb`, respectively. Then, `f_b` sets these bar values to $\mathtt{fb} \cdot \frac{\partial \mathtt{f}}{\partial \mathtt{u}}$ and $\mathtt{fb} \cdot \frac{\partial \mathtt{f}}{\partial \mathtt{v}}$, respectively, similar to the terms on the right hand side of (2.17).

The `while` loop in the code of `f_b` belongs to the forward pass; Tapenade has augmented the primal code (blue) with

- a statement that pushes the intermediate value `u` to the stack every time before it is overwritten (dark green), and
- statements that count the number of iterations, and push this number onto the stack as well (light green).

In the reverse pass code created by Tapenade (yellow), a `for` loop with the same number of iterations contains the bar value updates (2.17), (2.18) of the primal statement `x = x*y` in its loop body. To see this, rewrite the primal statement as

$$\mathtt{u}_{\text{old}} = \mathtt{u}, \quad \text{then} \tag{2.20}$$

$$\mathtt{u}_{\text{new}} = \mathtt{u}_{\text{old}} \cdot \mathtt{v}, \quad \text{then} \tag{2.21}$$

$$\mathtt{u} = \mathtt{u}_{\text{new}} \tag{2.22}$$

Listing 2.3: The code on the right was produced from the code on the left using the reverse mode of the source-rewriting AD tool Tapenade 3.16. Tapenade augmented the primal code (blue) with iteration counting (light green), storing of intermediate values (dark green) and the reverse pass (yellow).

```
double f(double u, double v){
  while(u<100){
    u = u*v;
  }
  return u;
}
```

```
#include <adStack.h>

void f_b(double u, double *ub, double↩
     v, double *vb, double fb) {
    double f;
    int adCount;
    int i;
    adCount = 0;
    while(u < 100) {
        pushReal8(u);
        u = u*v;
        adCount = adCount + 1;
    }
    pushInteger4(adCount);
    *ub = fb;
    *vb = 0.0;
    popInteger4(&adCount);
    for (i = 1; i < adCount+1; ++i) {
        popReal8(&u);
        *vb = *vb + u*(*ub);
        *ub = v*(*ub);
    }
}
```

with temporary variables $\mathtt{u}_{\text{old}}$, $\mathtt{u}_{\text{new}}$ that are used nowhere else. The respective bar value updates in reverse order read

$$\bar{\mathtt{u}}_{\text{new}} \mathrel{+}= \frac{\partial \bar{\mathtt{u}}_{\text{new}}}{\partial \bar{\mathtt{u}}_{\text{new}}} \cdot \bar{\mathtt{u}} = \bar{\mathtt{u}} \tag{2.23}$$

$$\bar{\mathtt{u}} = 0, \quad \text{then} \tag{2.24}$$

$$\bar{\mathtt{u}}_{\text{old}} \mathrel{+}= \frac{\partial(\mathtt{u}_{\text{old}} \cdot \mathtt{v})}{\partial \mathtt{u}_{\text{old}}} \cdot \bar{\mathtt{u}}_{\text{new}} = \mathtt{v} \cdot \bar{\mathtt{u}}_{\text{new}} \tag{2.25}$$

$$\bar{\mathtt{v}} \mathrel{+}= \frac{\partial(\mathtt{u}_{\text{old}} \cdot \mathtt{v})}{\partial \mathtt{v}} \cdot \bar{\mathtt{u}}_{\text{new}} = \mathtt{u}_{\text{old}} \cdot \bar{\mathtt{u}}_{\text{new}} \tag{2.26}$$

$$\bar{\mathtt{u}}_{\text{new}} = 0, \quad \text{then} \tag{2.27}$$

$$\bar{\mathtt{u}} \mathrel{+}= \frac{\partial \bar{\mathtt{u}}}{\partial \bar{\mathtt{u}}} \cdot \bar{\mathtt{u}}_{\text{old}} = \bar{\mathtt{u}}_{\text{old}}, \tag{2.28}$$

$$\bar{\mathtt{u}}_{\text{old}} = 0, \tag{2.29}$$

Given the fact that $\bar{\mathtt{u}}_{\text{old}}$ and $\bar{\mathtt{u}}_{\text{new}}$ are zero before evaluating (2.23)–(2.29), and are discarded afterwards, we can shorten this block to

$$\bar{\mathtt{u}}_{\text{new}} = \bar{\mathtt{u}} \tag{2.30}$$

$$\bar{\mathtt{u}}_{\text{old}} = \mathtt{v} \cdot \bar{\mathtt{u}}_{\text{new}} \tag{2.31}$$

$$\bar{\mathtt{v}} \mathrel{+}= \mathtt{u}_{\text{old}} \cdot \bar{\mathtt{u}}_{\text{new}} \tag{2.32}$$

$$\bar{\mathtt{u}} = \bar{\mathtt{u}}_{\text{old}}. \tag{2.33}$$

This can be simplified to

$$\bar{\mathtt{v}} \mathrel{+}= \mathtt{u}_{\text{old}} \cdot \bar{\mathtt{u}} \tag{2.34}$$

$$\bar{\mathtt{u}} = \mathtt{v} \cdot \bar{\mathtt{u}}, \tag{2.35}$$

which is what the body of the `for` loop generated by Tapenade in Listing 2.3 computes.

The forward and reverse passes need not be clearly separated in the differentiated code. In fact, they do not even need to run one after another: When we add the code for the function `g` in the left box of Listing 2.4 and ask Tapenade for its derivative, Tapenade additionally emits the code in the right box of Listing 2.4 (plus a function `f_c` with the same semantic as `f`). In each of the two calls to `f_b`, a forward pass with pushes onto the stack is followed by a reverse pass with evaluations of (2.17) taking data from the stack.

Besides Tapenade, other AD tools with a source-rewriting forward pass include Enzyme[128] and Clad[179].

2.4.5. Tape-Recording Approach

As an alternative to parsing the source code and transforming control flow constructs, a *tape-recording* forward pass records the entire stream of real-arithmetic statements on the forward-pass data structure called the *tape*. To this end, the primal program is augmented with the following *index handling* and *tape recording* logic.

Listing 2.4: The code on the right was produced from the code on the left using the reverse mode of the source transformation AD tool Tapenade. This example shows that the forward and reverse pass can be intertwined.

```
double f(double u, double v){
  while(u<100){
    u = u*v;
  }
  return u;
}

double g(double u1, double u2↩
    , double v1, double v2){
  return f(u1,v1) + f(u2,v2);
}
```

```
void g_b(double u1, double *u1b, ↩
    double u2, double *u2b, double v1↩
    , double *
        v1b, double v2, double *v2b, ↩
            double gb) {
    double result1;
    double result1b;
    double result2;
    double result2b;
    double g;
    result1b = gb;
    result2b = gb;
    *u2b = 0.0;
    *v2b = 0.0;
    f_b(u2, u2b, v2, v2b, result2b);
    *u1b = 0.0;
    *v1b = 0.0;
    f_b(u1, u1b, v1, v1b, result1b);
}
```

2.4.5.a. Index Handling. The index handling logic keeps track of an *index* of every number a, in order to identify it; in absence of a naming convention in the AD community, we will denote the index as $\hat{a}$ in this thesis. Typically, indices are chosen from the set $\{0, 1, 2, \dots\}$ of non-negative integers. A default index like 0 may be reserved for *passive* real numbers, which do not depend on any AD input. When a number is declared as an AD input, or when it appears as the left hand side of a statement like (2.14) that has at least one active variable on the right hand side, it is *active* and receives a new index. New indices can simply be assigned consecutively (1, 2, ...) by a *linear index management* strategy. Alternatively, if it is possible to track when an index is not used anymore because (all copies of) the corresponding variable disappeared, a *reuse index management*[159] strategy can reassign the index later on. Reusing indices saves space for the bar variables and makes it less likely that indices overflow a 32-bit integer. Either way, at any point of time, indices must uniquely identify a single node in the real-arithmetic evaluation tree; i. e., different numbers (in the sense of different variables, not values) shall be assigned different indices. Mere copies of a number may be allowed to carry the same index (*copy optimization*).

2.4.5.b. Tape Recording. For every statement (2.14) performed in the forward pass with at least one active operand, the tape recording logic stores information on the tape that allows to efficiently perform (2.17) and (2.18) later in the reverse pass.

In the *Jacobian taping* approach, the indices $\hat{a}_{\text{lhs}}$, $\hat{a}_1$, ..., $\hat{a}_k$, and the values of the partial derivatives $\frac{\partial\phi}{\partial a_1}(a_1, \dots, a_k)$, ..., $\frac{\partial\phi}{\partial a_k}(a_1, \dots, a_k)$ are stored on the tape.[158]

In the *primal value taping* approach, the indices $\hat{a}_{\text{lhs}}$, $\hat{a}_1$, ..., $\hat{a}_k$, the values of a_{lhs} and of any passive a_i, and a handle to a function to evaluate $\frac{\partial\phi}{\partial a_1}$, ..., $\frac{\partial\phi}{\partial a_k}$ are stored on the tape.[157]

The Jacobian taping approach seems easier to implement, and the primal value taping approach can save tape space if many statements have many arguments. For either approach, the index of the left hand side may be omitted if it can be reconstructed otherwise, e. g. from the position of the data on the tape.

2.4.5.c. Alternative Implementation of the Forward Mode. Besides the reverse-mode updates (2.17) and (2.18), the information on the tape can also be used to propagate dot values according to (2.15). When forward-mode derivatives are sought for many inputs as in Section 2.3.2, it might save run-time to record a tape once, and then evaluate it the corresponding number of times (possibly in a vector-mode fashion), compared to a repeated application of the forward mode, as the active real-arithmetic statements only contribute to a part of the run-time of the primal program.[73]

2.4.6. Asymptotical Performance

The run-time of the forward and reverse pass of reverse-mode AD with a single output variable is at most proportional to the run-time of the primal program, though the proportionality constants are usually higher than for forward-mode AD with a single input variable. The fact that the reverse-mode run-time is proportional to the number of AD outputs and constant in the number of AD inputs (Section 2.4.2), opposite to the forward mode (Section 2.3.2), makes reverse-mode AD much better suited for optimization problems with a single objective function value (AD output) depending on many parameters (AD inputs).

However, a price for the better run-time asymptotics has to be paid in terms of memory consumption of the forward-pass data structure. Especially for tape-recording forward passes (Section 2.4.5), the tape size is proportional to the number of active real-arithmetic statements, and can thus be expected to scale with the run-time of the primal program. Thus, limits on the amount of available memory translate to limits on the run-time of the primal program, if no further strategies to reduce tape size, such as those listed in the following, are applied.

Checkpoints[46,133] of the program state, taken at a few points of time during program execution, allow to record parts of the tape only shortly before they are needed for the reverse pass. *Preaccumulation*[11,178] replaces sections of a Jacobian tape by an equivalent multiplication with a local Jacobian; this saves space when the numbers of local inputs and outputs of the section are small. More complex AD workflows that respect the mathematical structure of the primal program, as introduced in Paragraph 2.2.d, usually also save tape space.

2.4.7. Comment on the Names

Throughout this section, we have used the terms *source-rewriting* and *tape-recording* for the two different approaches concerning which data is stored in the forward pass to allow for an efficient reverse pass. In the literature, authors sometimes prefer the terms *source-transformation* and *operator-overloading*, respectively. In this dissertation, we have reserved these terms for naming different mechanism to interact with the primal program in order to obtain knowledge on the real-arithmetic statements, in order to implement the forward mode, and either source-rewriting or tape-recording forward passes for reverse-mode AD. We review these mechanism next in Sections 2.5 to 2.7. Let us already state here that operator-overloading reverse-mode AD tools usually have a tape-recording forward pass. Source-transformation and compiler-based reverse-mode AD tools usually have a source-rewriting forward pass, although a tape-recording forward pass would also be possible.

2.5. Operator Overloading

2.5.1. New Floating-Point Type

A common feature of object-oriented programming languages is that new data types can be defined, specifying the data fields they contain and the methods they provide. *Operator-overloading* AD tools define a new type that should be used in the primal program instead of the built-in floating-point types. The new type stores a floating-point value and AD meta-data. It defines real-arithmetic operators and math functions that operate as expected on the value, and additionally perform (forward- or reverse-mode) AD logic. Listing 2.5 gives an idea of how a C++ type with additional dot-value propagation logic could look like.

As the example shows, C++ allows to re-define the methods `operator+`, `operator*`, ... that are called for operators like + or *; this is also possible in Python where they are called `__add__`, `__mul__`, ... In addition, C++ allows to define methods with existing function names when they have a distinctive signature, e. g. taking arguments of the new type; this allows to overload math functions like `sin` as well. Next to these, it makes sense to overload arithmetic assignment operators like `*=`, comparison operators like < and >=, type traits like `std::numeric_limits`, the operators << (with output streams) and >> (with input streams), etc.

2.5.2. "Automatic" except for the type exchange?

The closer the interface of the AD type resembles the interface of the built-in floating-point types, the easier it becomes to integrate AD into an existing codebase in a black-box fashion. Ideally, we would like the user to only have to do the following two things.

2.5.2.a. Automatic changes across the entire code. With statically typed languages like C++, the user has to exchange the floating-point type everywhere in the source code. This can be done automatically with an utility program like `sed`. Additionally, it might be necessary to add an `#include` (C++), `import` (Python) or similar statement in the beginning of each file, which can also be automated.

Listing 2.5: Prototypical operator-overloading forward-mode AD tool for C++. The type `ad_number` has a member variable for the `value` of the number, and overloads real-arithmetic operations and functions so that they are performed on the `value`. Additionally, an `ad_number` stores AD meta-data (here: dot values) and the overloads perform AD logic (here: propagation of dot values).

```cpp
// ad_number.h
#include <math.h>

struct ad_number {
  double value;
  double dot_value;

  ad_number(double v): value(v), dot_value(0.0) {}
  ad_number(): value(0.0), dot_value(0.0) {}

};

inline ad_number operator+(ad_number a, ad_number b){
  ad_number result(a.value + b.value);
  result.dot_value = a.dot_value + b.dot_value;
  return result;
}

inline ad_number operator*(ad_number a, ad_number b){
  ad_number result(a.value * b.value);
  result.dot_value = a.dot_value * b.value
                   + a.value * b.dot_value;
  return result;
}

inline ad_number sin(ad_number a){
  ad_number result(sin(a.value));
  result.dot_value = cos(a.value) * a.dot_value;
  return result;
}

/* ... define further operations ... */
```

Listing 2.6: C++ program differentiated using the prototypical operator-overloading forward-mode tool in Listing 2.5. Apart from the type exchange, we inserted code to include the AD tool (gray), declare AD inputs (light red), and declare AD outputs (dark red).

```
#include "ad_number.h"
#include <iostream>
#include <math.h>

ad_number func(ad_number a){
  return a * a + sin(a);
}

int main(){
  ad_number a;
  a.value = 4.0;
  a.dot_value = 1.0;
  ad_number b = func(a);
  std::cout
    << "b     = " << b.value << std::endl
    << "db/da= " << b.dot_value << std::endl;
}
```

2.5.2.b. Manual changes at specific locations only. Besides, the user has to edit the code to mark the AD inputs and outputs, possibly trigger the recording and derivative computation, and to utilize the derivatives. Examples for this are shown in Listing 2.6 for the prototypical forward-mode tool (Listing 2.5), and in Listing 2.7 for the reverse mode of the operator-overloading AD tool CoDiPack[158]. Clearly, these changes cannot be performed automatically, as the user provides information by adding these statements. However, the effort required by this manual intervention should roughly scale with the number of AD inputs and outputs rather than with the total size of the codebase.

In practice, additional manual modifications can be required for several reasons such as the following.

2.5.2.c. Type Exchange Mistakes. Exchanging the floating-point types is not as trivial as it sounds. When the strings `long double`, `double` and `float` are substituted by an AD type specifier in all C++ source and header files, this might damage e. g. string literals, names of `#include`'d files such as the standard library headers `float.h` and `cfloat`, or names of variables, functions, macros or types in external, non-differentiated libraries, like `std::hexfloat` in the standard library header `ios`. When multiple floating-point types are mapped to a single AD type, legal overloads for these types might become illegal function redefinitions. Listing 2.8 demonstrates that floating-point literals may need to be converted to the AD type as well. Header files provided by the AD tool to declare the AD type must be included before the first use of the AD type in all translation units; simply prepending every file that contains source code with such an include statement can lead to bugs if some of these files are not included in the global scope.

Listing 2.7: C++ program differentiated using the reverse mode of the operator-overloading tool CoDiPack version 2.1.0. We inserted code to include the AD tool (gray), declare AD inputs (light red) and AD outputs (dark red) during the forward pass, control the tape recording (green) and perform a reverse pass (yellow).

```
#include "codi.hpp"
#include <iostream>
#include <math.h>

using Real = codi::RealReverse;

Real func(Real a){
  return a * a + sin(a);
}

int main(){
  auto& tape = Real::getTape();
  tape.setActive();
  Real a = 4.0;
  tape.registerInput(a);
  Real b = func(a);
  tape.registerOutput(b);
  tape.setPassive();
  b.setGradient(1.0);
  tape.evaluate();
  std::cout
    << "b     = " << b << std::endl
    << "db/da= " << a.getGradient() << std::endl;
}
```

Listing 2.8: C++ program where a simple type exchange leads to a compiler error. The function template `f` requires two arguments of the same type, so the floating-point literal would need to be converted to `ad_number` like the first argument.

```cpp
#include "ad_number.h"

template<typename T>
T f(T a, T b){
  return a+b;
}

int main(){
  ad_number a = 1;
  f(a, 1.0);
}
```

```
test.cpp: In function ‘int main()’:
test.cpp:10:4: error: no matching function for call to ‘f(ad_number&, ↩
    double)’
   10 |   f(a, 1.0);
      |   ~^~~~~~~~~
test.cpp:4:3: note: candidate: ‘template<class T> T f(T, T)’
    4 | T f(T a, T b){
      |   ^
test.cpp:4:3: note:   template argument deduction/substitution failed:
test.cpp:10:4: note:   deduced conflicting types for parameter ‘T’ (‘↩
    ad_number’ and ‘double’)
   10 |   f(a, 1.0);
      |   ~^~~~~~~~~
```

2.5.2.d. External Libraries. Calls to functions from external libraries break if the type of an argument changes. If the library call does not contribute to the differentiable evaluation tree, like dumping data into files or `printf`-like logging, it suffices to extract the floating-point value from the AD type for the library function call. This extraction could be accomplished automatically at some places of the code if the AD type defines an implicit cast to `double`, but this comes with its own set of problems (see Paragraph 2.5.2.f below). In any case, the compiler will not use implicit casts if a pointer to an AD type array needs to be converted to a pointer to a `double` array, a non-const reference to an AD type variable is to be converted to `double&`, when AD type arguments are passed to `printf`, etc.

If the library call contributes to the differentiable evaluation tree, then either the type exchange has to be applied to the library as well, or the library functions must be wrapped to ensure correct AD handling. For instance, if a parallel program uses the Message Passing Interface (MPI) to communicate floating-point data between processes, the MPI library needs to be wrapped by the AD tool.

2.5.2.e. Type Conversions. Listing 2.8 demonstrates that a simple type exchange can break templated code due to the lack of type conversions during template argument deduction. Another potential problem related to type conversions is that C++ does not implicitly perform more than one user-defined type conversion at a time. For example, the C++ code in Listing 2.9 can be compiled with `g++ -c` without problems; note that the `double` literal `3.14` can be passed as a `C<double>` argument of `do_something` because the converting constructor is implicitly called. But when the type `number` is changed from `double` to `ad_number`, the program is ill-formed because the converting constructor from `double` to `ad_number` must be called in addition to the converting constructor from `ad_number` to `C<ad_number>`. The error message of GCC 11.4.0 is shown in Listing 2.9, along with the analogous error message when CoDiPack's `codi::RealForward` type is used.

2.5.2.f. Benefits and Issues of Implicit Casts to `int`/`double`. A related issue comes up with respect to integer casts and C++'s ternary `?:` operator. An AD type `D` typically supports implicit casts `double` → `D` so literals of type `double`, `int` etc. can be used mostly in the same way as before (with exceptions like Listings 2.8 and 2.9). To give `D` a behavior as close to `double` as possible, one could also think of defining an implicit cast `D` → `int`, so assignments from `D` to integers work. By the latter cast and the implicit conversion `int` → `double`, it becomes possible to convert `D` → `double`. But then, the second and third operand of a conditional operator expression like `true ? (D)1.0 : 2.0` can be converted to one another, which makes the program ill-formed;[94] naturally, as it is not clear whether the type of the conditional expression should be `D` or `double`. This may look very specific, but it was one of the issues that we faced while trying to apply operator-overloading AD to the high-energy-physics simulation package G4HepEm[142]/HepEmShow[141] (see Section 10.4), which frequently uses assignments to integers for rounding, as well as the `?:` operator.

Listing 2.9: When the type `number` is changed from the built-in `double` to the user-defined AD type `ad_number` of Listing 2.5, or CoDiPack's `codi::RealForward` type, this valid C++ program becomes ill-formed. GCC 11.4.0 produces the error messages shown below. An implicit conversion from the `double` literal `3.14` to `C<ad_number>` or `C<codi::RealForward>` is not possible, because it would require two user-defined conversion operations.

```
#include "ad_number.h"
using number = double; // replacing double -> ad_number gives compiler error

template<typename T>
struct C {
  T x;
  C() {}
  C(T x): x(x) {}
};
void do_something(C<number> c) {}

int main(){
  do_something(3.14);
}
```

```
example.cpp: In function 'int main()':
example.cpp:13:16: error: could not convert '3.1400000000000001e+0' from 'double↩
    ' to 'C<ad_number>'
   13 |   do_something(3.14);
      |                ^~~~
      |                |
      |                double
```

```
example.cpp: In function 'int main()':
example.cpp:13:16: error: could not convert '3.1400000000000001e+0' from 'double↩
    ' to 'C<codi::ActiveType<codi::ForwardEvaluation<double, double> > >'
   13 |   do_something(3.14);
      |                ^~~~
      |                |
      |                double
```

Listing 2.10: Changing the type `number` from `double` to CoDiPack's `codi::RealForward` type affects the behavior of this program, printing `f called` twice instead of once. This is because overloads of the `&&` operator do not use short-circuit evaluation (Paragraph 2.5.2.g).

```
#include "codi.hpp"
#include <iostream>

using number = double; // replacing double -> codi::RealForward changes output

number f(number x){
  std::cout << "f called\n";
  return x;
}

int main(){
  number a=0, b=1;
  f(a) && f(b);
}
```

Listing 2.11: A simple type exchange from `double` to `ad_number` (see Listing 2.5) in this C++ function is not sufficient to obtain correct derivatives.

```
double func(double a){
  unsigned long long* p = (unsigned long long*)&a;
  *p ^= 0x8000000000000000ul;
  return a;
}
```

Apart from such specific types of issues, having a cast `D` $\rightarrow$ `double` also creates the more abstract danger that C++ resolves other incompatibilities between `D` and `double` by using this cast, and thus discarding AD information, at unexpected locations.

2.5.2.g. Other Differences in the Standard. The C++ standards[94–97] sometimes make other subtle distinctions between built-in and class types. For instance, the built-in logical operators `&&` and `||` in C++ do not evaluate their second argument if the first argument already determines the result (i. e. if it is `false` for `&&`, or `true` in `||`), which is called *short-circuit evaluation*. Their overloads, however, lose this property, as shown in Listing 2.10.

As a different example, the template `std::complex<T>` from the standard header `<complex>` implements complex floating-point arithmetic if the type `T` is `float`, `double`, or `long double`, but its behavior is unspecified if `T` is any other type, such as an AD type. Concerning type traits like

`std::is_arithmetic<T>`, `std::is_floating_point<T>`, `std::is_class<T>`,

their whole point is to behave differently if `T` is e. g. `double` vs. an AD class type.

Some more examples have been collected by Hück et al.[88]

2.5.2.h. Assumptions in the Primal Program. If the primal program has hard-coded assumptions on the size or internal structure of the floating-point type, these may be violated by a type exchange. For instance, for common implementations of the programming language C, the function in Listing 2.11 returns the negative of its argument: The `^=` operator is used to apply a bitwise logical "exclusive-or" to the argument, and the other operand `0x80...0` contains a single 1-bit at the position where the sign of a `double` is usually stored, thus flipping the sign. As a side note, the binary representation of floating-point numbers is unspecified in C and implementation-defined in C++ according to the respective standards,[92–97] but the most popular choice is the IEEE-754 formats[90] discussed in Section 3.1 where the previous statement is true. After a naive type exchange that replaces every occurrence of `double` with the `ad_number` type defined in Listing 2.5, the function would flip the sign of the `value` but leave the sign of the `dot_value` untouched. The same happens with CoDiPack's `RealForward` type. If we had decided in Listing 2.5 to store the `dot_value` before the `value`, the former would be negated but the latter would remain constant.

2.5.3. Available Tools

In our perception, operator overloading is the most popular mechanism to discover and augment floating-point operations. Just naming a few operator-overloading AD tools, ADOL-C[184] and CoDiPack[158] are applicable to C++ codes; an example for using the reverse mode of CoDiPack is shown in Listing 2.7. The autograd tool[123] used by PyTorch[146], and the internal AD tool of TensorFlow[1], provide derivatives of calculations made by Python code and have operator-overloading-type front-ends. Their main intention is to provide derivatives through neural networks, but see Listing 2.12 for a more generic example.

2.6. Source Transformation

Source transformation AD tools like TAPENADE[79] and ADIFOR[27,29] parse the source code of the primal program to learn about the real-arithmetic operations it performs, and emit new source with additional AD logic. For forward-mode AD, source transformation tools may simply insert the dot-value-propagating AD logic in the respective programming language; Listing 2.13 shows an example of this. In the same style, source transformation tools could insert forward-pass logic for a tape-recording reverse mode into the code; however, Tapenade follows the source-rewriting approach. See Section 2.4.4 for a detailed example of the code produced by Tapenade.

When applied to the function in Listing 2.11, which negates a number by flipping the sign bit (in many implementations of C but outside of the language standard), Tapenade does not see a differentiable dependency. This behaviour is justified, as Tapenade has no control about the floating-point representation used by the compiler, and therefore cannot even know what kind of real arithmetic is performed here. If correct handling of this bit-trick were desired, we presume that it would be possible to implement it in Tapenade. If the other operand to the bit-wise operation is not statically defined (i. e. as a

Listing 2.12: Program differentiated in *reverse mode* AD with PyTorch[146] (top) and TensorFlow[1] (bottom).

```python
import torch

def func(a):
  return a*a + torch.sin(a)

a = torch.tensor(2.,
                requires_grad=True)
b = func(a)

b.backward()
print("b    =", b.item())
print("db/da=", a.grad)
# alternative:
# torch.autograd.grad(b, [a])
```

```python
import tensorflow as tf

def func(a):
  return a*a + tf.math.sin(a)

a = tf.Variable(2.)
with tf.GradientTape() as tape:
  b = func(a)

print("b    =", b)
print("db/da=", tape.gradient(b,a))
```

Listing 2.13: The code on the right was produced from the code on the left using the forward mode of the source transformation AD tool Tapenade 3.16.

```c
double f(double u, double v){
  while(u<100){
    u = u*v;
  }
  return u;
}
```

```c
double f_d(double u, double ud, ↩
    double v, double vd, double *f↩
    ) {
     while(u < 100) {
         ud = v*ud + u*vd;
         u = u*v;
     }
     *f = u;
     return ud;
}
```

literal `0x80...0` in the code) but rather acquires its value at run-time (e. g. a non-constant global variable initialized at run-time), source-rewriting tools cannot discover them at compile-time and would need to insert run-time checks. We pay attention to this detail here in order to indicate that bit-tricks (as in Listing 2.11 and later in Section 3.3), which are only partially supported by Derivgrind, can also be a problem for source-code-based AD tools if they appear on the source code level (which can happen in practice, e. g. in Geant4, as outlined in Paragraph 10.3.e).

Tapenade can differentiate primal programs written in Fortran 77, Fortran 95, and C. As source-transformation AD tools have to parse the entire control flow structure of the primal program, supporting languages with a more complex syntax, such as C++, is considered difficult. Several source-rewriting AD tools have solved this problem by making use of existing compiler infrastructure, as we will see in the next Section 2.7.

2.7. Compiler-Based Tools

AD logic can also be added during compilation of the primal program. To give a brief overview here, we will look at the two AD tools Clad[179] and Enzyme[128], which both use the LLVM compiler infrastructure[110,117] but operate at different stages of the compilation process. Clad is a plugin for the Clang C/C++ compiler front-end, operating on its internal representation of the source code in the form of an abstract syntax tree (AST). Enzyme[128] operates like an optimization pass on the LLVM internal representation (IR) produced by LLVM front-ends. As LLVM has front-ends for various programming languages (such as C, C++, Julia, Rust, ...), Enzyme can be considered the most cross-language tool so far. Even graphics processing unit (GPU) kernels can be differentiated with Enzyme.[129]

Clad has an option to output the differentiated code as C++ source code, which we use in Listing 2.14 for the same function as our reverse-mode Tapenade example in Listing 2.3, obtaining structurally similar differentiated code.

Enzyme handles the "negation by flipping the sign bit" trick in Listing 2.11 correctly. When the operand `0x80...0` is stored in a non-constant global variable, it cannot handle it any more (and gives an error message), as generally anticipated for source-rewriting tools in the previous Section 2.6.

In the reverse mode, at the moment, both Clad and Enzyme struggle with global variables: For the code shown in Listing 2.15, they both compute a derivative of zero. Enzyme requires that global variables are explicitly annotated in the LLVM IR.

2.8. Novel Approach: Machine-Code-Based Tool

The source transformation (Section 2.6), operator overloading (Section 2.5) and compiler-based (Section 2.7) approaches operate from an increasing distance to the source code of the primal program: While source transformation tools parse the source code, operator overloading tools (ideally) require only a simple type exchange, and compiler-based tools

Listing 2.14: The code on the bottom was produced from the code on the top using the reverse mode of the compiler-based source-rewriting AD tool Clad. Compare with the Tapenade output in Listing 2.3.

```
#include "clad/Differentiator/Differentiator.h"

double f(double u, double v){
  while(u<100){
    u = u*v;
  }
  return u;
}

int main() {
  auto df = clad::gradient(f, "u,v");
  df.dump();
}
```

```
void f_grad(double u, double v, clad::array_ref<double> _d_u, clad::↩
    array_ref<double> _d_v) {
    unsigned long _t0;
    clad::tape<double> _t1 = {};
    _t0 = 0;
    while (u < 100)
        {
            _t0++;
            clad::push(_t1, u);
            u = u * v;
        }
    goto _label0;
  _label0:
    * _d_u += 1;
    while (_t0)
        {
            {
                {
                    u = clad::pop(_t1);
                    double _r_d0 = * _d_u;
                    * _d_u += _r_d0 * v;
                    * _d_v += u * _r_d0;
                    * _d_u -= _r_d0;
                    * _d_u;
                }
            }
            _t0--;
        }
}
```

Listing 2.15: Without further manual adaptations, functions involving global variables are not properly differentiated by Clad (top) and Enzyme (bottom). Both AD tools compute a zero value for the derivative which should be `6.0`.

```
#include <iostream>
#include "clad/Differentiator/Differentiator.h"

double g;

double f(double u){
  g = u;
  return g*g;
}

int main() {
  auto df = clad::gradient(f, "u");
  double du=0;
  df.execute(3, &du);
  std::cout << du << std::endl;
}
```

```
#include <iostream>

double g;

double f(double u){
  g = u;
  return g*g;
}

extern double __enzyme_autodiff(void*, double);

int main(int argc, char* argv[]) {
  std::cout << __enzyme_autodiff((void*)f, 3.0) << std::endl;
}
```

only access internal compiler representations of the code. The Enzyme tool does not even require source code of static libraries, if they contain LLVM IR bitcode.

In this dissertation, we continue this journey away from the source code. Our AD tool *Derivgrind* employs a dynamic binary instrumentation framework (see Chapter 4) to insert AD logic into the machine code (Chapter 3) of the primal program just before it executes on the processor. In Sections 2.8.1 and 2.8.2, we give an abstract overview of the chances and limitations of machine-code-based AD, based on the competing design goals listed in Paragraph 1.1.c.

2.8.1. Advantages of Machine-Code-Based AD

2.8.1.a. Scope of Supported Primal Programs. Machine-code-based AD supports *cross-language* and, to some extent, *partially closed-source* codebases.

A large variety of programming languages exist to write computer programs, and a single computer program may have many components written in different languages. When a single source-code-based AD tool is to be applied to an existing codebase, it must support all languages from which contributions to the real-arithmetic evaluation tree are made. Given the huge differences that exist among popular programming languages, creating a common AD tool for all of them can be considered practically impossible, and it would be better to invest the effort of combining multiple AD tools, which likely requires manual adaptations of interfaces in the codebase.

Enzyme supports a larger variety of programming languages, by relying on the LLVM front-ends that translate these programming languages into the common LLVM IR language. However, the eventual common language, into which all programming languages finally translate, is the machine code executing on the processor. Machine-code-based AD can operate on machine code originating from a wild mixture of obscure programming languages in the same way as it operates on a compiled C++ program.[69]

Sometimes, software is only distributed in compiled form without the source code. If software projects use closed-source components for real arithmetic with relevance for the derivative, source-code-based AD is not an option any more. Enzyme can deal with closed-source static libraries if they include their LLVM IR. For machine-code-based AD, it does not make a difference whether the source code of intermediate real-arithmetic calculations is available or not. (Only if things go wrong, having the source code is good for debugging, and potentially required to fix bit-tricks, as discussed below.)

While machine-code-based AD can thus insert AD logic into a much wider range of programs than source-code-based AD tools, there are limits when it comes to how AD inputs and outputs are identified. The most common way for that is to refer to the source code by a file name, line number and variable name; naturally, even machine-code-based AD tools need some kind of access to the source code to make use of such specifications. The scope of supported primal programs is further limited by the fact that it is at least very difficult (and likely impossible) to comprehensively cover all possible "bit-tricks" similar to Listings 2.2 and 2.11. Finally, software licenses may impose legal constraints on whether certain primal programs may be disassembled and modified.

2.8.1.b. Simple Integration of AD into Primal Code. The source transformation and operator overloading approaches create or modify basically every section of the source code that contributes to the real-arithmetic evaluation tree; this is mostly automatic, but not entirely. In contrast, machine-code-based AD does not require any interaction with the source code and build system, except for what is needed to identify AD inputs and outputs, and to identify and fix any bit-tricks. It may thus reduce the amount of effort required from the AD tool user to differentiate a given primal program in a "black-box" fashion.

2.8.2. Limitations of Machine-Code-Based AD

2.8.2.a. Correctness in View of Bit-Tricks. With Listing 2.11, we have outlined one example for the fact that on the level of bits and bytes, real-arithmetic operations can sometimes be represented with integer-arithmetic or bitwise logical operations. Listing 2.2 shows another construct that AD tools will likely not recognize as relevant for the derivative computation. In Section 3.3, we list a few more examples of such *bit-tricks*, though they are not systematically understood. As AD depends on the full real-arithmetic evaluation tree, AD tools might give wrong results for primal programs using bit-tricks that are not recognized and handled correctly by the AD tool.

As far as bit-tricks can be explicitly implemented in the source code of the primal program, both source-code and machine-code-based AD tools have to deal with them. Source code and language standards may however provide more information than pure machine code, so it might be harder for machine-code-based AD tools to recognize bit-tricks. Additionally, bit-tricks introduced by the compiler, e. g. because of performance reasons, can become a problem solely for machine-code-based tools. Part of this thesis is to assess how much of a limitation this is in practice.

2.8.2.b. Applicability of Advanced AD Workflows. Knowledge on, and access to, the internal structure of the primal program is required in order to apply advanced AD techniques to reduce the tape size (Section 2.4.6) or to respect the mathematical structure of numerical algorithms (Paragraph 2.2.d). As the main use cases of machine-code-based AD evolve around enabling AD in exploratory or complex setups rather than improving AD performance, there has been no need to make advanced AD workflows available in Derivgrind so far.

2.8.2.c. Performance. Compilers are very good at producing performant machine code that makes efficient use of the processor. As code optimization passes can be run after source-code-based tools, the resulting machine code is usually performant. Retrospectively inserting AD logic into optimized machine code, on the other hand, is likely to destroy any performance optimizations.

For our novel tool Derivgrind, we have measured a much larger AD slow-down than for CoDiPack (Chapter 9). Derivgrind relies on the Valgrind framework to interact with the primal program; most other tools implemented in the Valgrind framework serve debugging, profiling, or security purposes. These are usually not performance-critical, and performance has been a subordinate objective during the development of Derivgrind, too. Thus, there might be potential for better performance in machine-code-based AD. For now, reduced performance is a price that we are happy to pay in exchange for universal applicability requiring little manual efforts.

3. Compiled Programs

The goal of this thesis is to apply algorithmic differentiation (Chapter 2) to compiled computer programs. In this chapter, we will therefore explore a bit of the low-level internals of computer software, focussing on how software performs real arithmetic.

The set $\mathbb{R}$ of real numbers is uncountable, but a digital computer can only assume a finite number of states. Thus, real numbers cannot be accurately represented in a computer. Instead, *floating-point formats* define approximative represenions of real numbers as digital data, and we review the most important of these formats in Section 3.1.

As the core component of a computer, a *processor* (also called *central processing unit*, *CPU*) can transfer data between its *registers* and *main memory*, and perform a set of integer-arithmetic, floating-point-arithmetic, logic, and possibly more specialised operations on the data. All these actions are directed by machine-code *instructions* read from memory, according to the *von Neumann model.*[182] An *instruction set architecture* (ISA) is a comprehensive description of how a CPU behaves functionally, including the set of machine-code instructions, their binary encoding and semantic meaning. A single ISA can be implemented in hardware in a variety of ways; these are called *microarchitectures*, and can differ in their internal design and, e. g., the resulting performance characteristics. ISAs can also be implemented by emulator software. ISAs have often been extended over time to accomodate technological advancements.

At the time of writing this thesis (November 2023), a large majority of the top 500 most powerful computing clusters are based on processors from the x86-64 family of ISAs (besides *graphics processing unit (GPU)* accelerators omitted in this thesis), and all of them run an operating system based on the Linux kernel. In the perception of the author, this platform is also the most common target of scientific computing software, and it is thus the platform that this dissertation is focused on. In Section 3.2, we have a closer look at a selection of short snippets of assembly code compiled for x86-64 Linux systems, to understand how compiled programs look like, and to introduce important instructions and concepts.

Floating-point arithmetic can be performed by instructions other than the semantically correct floating-point instruction. As AD relies on a complete knowledge of the real-arithmetic evaluation tree of a program, it is mandatory to be aware of such “bit-tricks”, and we list a variety of them in Section 3.3. Throughout the remainder of this thesis, we will come back to the examples presented in this section.

3.1. Floating-Point Formats

While there is a canonical way of using blocks of several bytes to represent integers, early-day computer software used various formats to represent real numbers as digital

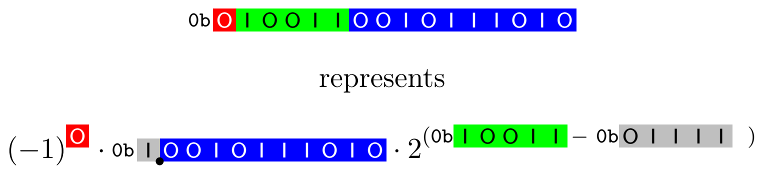

0b 0 1 0 0 1 1 0 0 1 0 1 1 1 0 1 0
represents
(−1)^0 · 0b 1.0 0 1 0 1 1 1 0 1 0 · 2^(0b 1 0 0 1 1 − 0b 0 1 1 1 1)

Figure 3.1: Structure of IEEE-754 binary floating-point formats: Sign bit (red), exponent bits (green) with bias (gray number, value determined by the standard), and mantissa/significand (blue) with implicit leading digit 1 (gray) and decimal point. The figure illustrates the rarely used `binary16` format; see Paragraphs 3.1.a to 3.1.c for details on `binary64` and `binary32`.

data. Basically three binary floating-point formats are still in widespread use today and directly supported by state-of-the-art ISAs. The IEEE 754 standard[90], published in 1985 and revised in 2008 and 2019, specifies the binary floating-point interchange formats `binary32` (before 2008: *single*) and `binary64` (before 2008: *double*) as described in the following, and illustrated in Figure 3.1.

3.1.a. 64-Bit Precision, Normal Case. In the 8-byte format `binary64`,

- the most significant bit stores the sign, `0` indicating a positive and `1` indicating a negative number,
- the 11 next-most significant bits store the integer exponent in a biased fashion, meaning that `0b00...01` and `0b11...10` represent the lowest and highest possible exponents −1022 and 1023, respectively, and
- the remaining 52 lower-significant bits store the significand (also called mantissa), apart from its implicit leading digit 1.

Thus, `0b`$b_{63}b_{62}\dots b_1b_0$ represents the real number

$$(-1)^{b_{63}} \cdot \left(1 \cdot 2^E + b_{51} \cdot 2^{E-1} + b_{50} \cdot 2^{E-2} + \dots + b_0 \cdot 2^{E-52}\right) \qquad (3.1)$$
$$\text{with} \quad E = b_{62} \cdot 2^{10} + b_{61} \cdot 2^9 + \dots + b_{52} \cdot 2^0 - 1023$$

if $E \neq -1023, 1024$. The value of the bias 1023 is specified in the IEEE-754 standard.

3.1.b. 64-Bit Precision, Sub-Normal, Infinite and Not-Numbers. In the case $E = -1023$ of all-zero exponent bits, a different formula

$$(-1)^{b_{63}} \cdot \left(b_{51} \cdot 2^E + b_{50} \cdot 2^{E-1} + \dots + b_0 \cdot 2^{E-51}\right) \qquad (3.2)$$

without an implicit leading digit 1 applies, to represent *sub-normal* numbers close to zero, with reduced accuracy. In particular, a 64-bit `0x00...00` is interpreted as +0.0. The case $E = 1024$ of all-one exponent bits is used to represent infinite numbers if all significand bits are set to zero, with the sign bit distinguishing between $+\infty$ and $-\infty$. When there are non-zero significand bits, the binary data represents not-numbers (NaNs) which might arise from, e. g., $(\pm 0)/(\pm 0)$, $(\pm 0) \cdot (\pm\infty)$, $\infty - \infty$, $\sqrt{a}$ and $\ln a$ for $a < 0$, or operations with NaN operands.

3.1.c. 32-Bit Precision. The 4-byte format `binary32` is defined in an analogous fashion with 8 exponent and 23 significand bits (apart from the leading 1), and exponents ranging from -126 to 127.

Besides `binary32`, `binary64`, and other floating-point formats, and floating-point arithmetic operations applicable to these, IEEE 754 also specifies *rounding modes* and *exceptions.*

3.1.d. Rounding Modes. When the infinitely precise result of an arithmetic operation cannot be represented the target floating-point format, it can be approximated in either of the following IEEE 754 rounding modes:

- *roundTiesToEven*, choosing the closest number and resolving ties in such a way that the last binary digit becomes 0,
- *roundTiesToAway*, choosing the closest number and resolving ties in favor of a larger absolute value,
- *roundTowardPositive*, choosing the closest number that is not less,
- *roundTowardNegative*, choosing the closest number that is not greater,
- *roundTowardZero*, choosing the closest number whose absolute value is not greater.

3.1.e. Exceptions. Floating-point operations typically raise an *invalid operation* exception if the result would be NaN. The *division by zero* exception is also raised when computing the logarithm of zero. Furthermore, there are exceptions raised when an *overflow* or *underflow* occurs or when the result has to be rounded (*inexact*).

3.1.f. 80-Bit Precision. Besides the IEEE-754 `binary64` and `binary32` formats, the 10-byte *x87 double extended precision* format is somewhat widespread. First implemented in the Intel 8087 coprocessor announced in 1980 and still used by x86-64 CPUs nowadays, it uses 15 exponent and 64 significand bits, including the explicit leading digit.[145]

3.1.g. Implementation of Floating-Point Operations. In the overview on the x86-64 ISA in the next Section 3.2, we will encounter floating-point registers and instructions. These instructions expect floating-point data to be represented in one of these three floating-point formats, and are typically implemented by dedicated circuitry. Yet there are more ways to digitally represent, and operate on, real numbers, which we discuss in Section 3.3.

3.2. The X86-64 Architecture and the Linux Kernel

This section gives an overview on the x86-64 ISAs with a particular focus on its floating-point extensions, as well as some higher-level concepts on a conventional GNU/Linux system. We have put a particular emphasis on instructions and concepts required to understand the later parts of this thesis, and present them through a sequence of assembly code snippets. See the specifications[15,125] for detailed and comprehensive information, and textbooks[147,172] for structured bottom-up introductions.

3.2.1. Assembly Code Basics

3.2.1.a. Instructions. Machine code for x86-64 processors is a sequence of binary encoded instructions of variable length. An instruction typically starts with an *opcode* that specifies the operation to be performed. One or several *prefixes* may precede the opcode and modify the operation. Subsequently, *operands* identify either the value or the memory location of the data, by constants or by register contents. The number of operands and the possible combinations of these *addressing modes* may depend on the operation. Binary machine code is usually represented by textual *assembly language* easier to read for humans. For example, assembly language replaces (prefixed) opcodes like `f3 0f 1e fa` by textual *mnemonics* like `endbr64`.

3.2.1.b. Generation of the Snippets. We will walk through a sequence of snippets of *assembly code* following the AT&T syntax, produced by the C compiler `gcc` from the GNU Compiler Collection (GCC) in version 11.3.0 with the flags `-S` (compile to assembly), `-O3` (optimize the code) and `-march=skylake` (specifies the target processor) on a Linux system, unless indicated otherwise. The snippets shown here are contiguous pieces of the compiler output, except that labels and directives have been removed when they were irrelevant for the demonstration.

3.2.1.c. Relation to Machine Code Running on the CPU. Before the assembly code can execute on the CPU, it is

1. translated to machine code, also called *object code* at this stage, by an *assembler*,
2. transformed and combined with other pieces of object code by a *linker*, to create an *executable* or *library*,
3. which undergoes another set of transformations when loaded into the main memory by a *loader* at the start-up or during the run-time of a program.

While the forward-mode AD prototype by Gendler et al.[69] operates on the assembly code (i. e. acts before these transformations take place), our AD tool *Derivgrind* inserts AD logic into the machine code just before it has a chance to execute on the CPU (i. e. after these transformations). The transformations affect e. g. hard-coded references in the machine code and may even change instructions (e. g. with link-time optimization).

Nevertheless, assembly code is a good approximation of the code that will run on the CPU, which is why we use it for our examples.

3.2.1.d. Calling Conventions. The concept of calling functions, methods, subroutines etc. might, at first, seem like a characteristic of higher-level programming languages like C. However, this concept is actually rooted in the machine code level: The *instruction pointer register* RIP of the processor stores the memory location of the next instruction to be executed, and storing another address by means of a jump or call instruction can realize a function call. *Calling conventions* specify the details of the "contract" between caller and callee, including

- the layout of a *call stack* that stores the return address and local variables of the callee, and how the work of growing the stack is distributed between caller and callee;
- how and in which order function parameters and return values are passed, e. g. on the call stack or in registers; and
- which registers must be preserved, and which may be overwritten, by the callee.

Linux systems on x86-64 stick to the System V ABI[125] calling convention.

3.2.1.e. Example for a Function Call. In our first snippet in Listing 3.1, the C code on the left side declares functions `f` and `g`, but only defines `g`, so the compiler only produces assembly code for `g`. The directive `.text` tells the assembler that what follows belongs to the *text segment*, i. e. the executable instructions part of the object file. The label `g:` instructs the assembler to remember the present code location (i. e. the location of the `endbr64` instruction), so it can be referred to from elsewhere, e. g. for function calls; the symbol `g` will also be visible in later linking stages because of the `.globl` directive. We have removed most of the directives in the following unless they are required to understand the assembly code or to infer the corresponding bytes of object code produced by the assembler.

The code of `g` starts with an `endbr64` instruction; this instruction does nothing but indicate a valid jump target address for Intel's Control-Flow Enforcement Technology (CET, 2016).[169] The following instructions manipulate the content of several registers. Operands starting with a dollar sign indicate integer literals, and operands starting with a percent sign indicate the content of register. The x86-64 architecture features sixteen general-purpose 64-bit registers RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8, R9, . . . , R15. The RSP pointer is generally used as a pointer to the top of the call stack, on which local variables and return addresses are stored. The `call` instruction performs several actions: It pushes the address of the next instruction, inferred from the special-purpose register RIP, to the stack, and it writes to RIP to perform the jump. The jump target address is not known at this stage, but it will be edited in the process of linking the program and loading it into memory, so that when execution reaches the `call` instruction, it points to code for `f` (in general).

Listing 3.1: Machine code of a function call with integer arguments.

```c
long f(long a, long b,
       long c, long d);

long g(){
  return f(-10,-20,30,40)+50;
}
```

```
  .text
  .p2align 4
  .globl  g
  .type g, @function
g:
  endbr64
  subq   $8, %rsp
  movl   $40, %ecx
  movl   $30, %edx
  movq   $-20, %rsi
  movq   $-10, %rdi
  call   f@PLT
  addq   $50, %rax
  addq   $8, %rsp
  ret
```

Listing 3.2: Pointer-type function arguments and indirect addressing.

```c
void f(long long* ptr){
  *ptr += 100;
}
```

```
f:
  endbr64
  addq   $100, (%rdi)
  ret
```

Opposite to the `call` instruction, `ret` pops a return address from the stack and jumps there. The `subq` and `addq` instructions applied to RSP, performing a 64-bit integer subtraction or addition, let the stack grow and shrink by eight bytes, respectively.

The first `movl` instruction in Listing 3.1 copies a 32-bit value `40` into the ECX register, which is the lower 32-bit half of the RCX register. The two `movl` and two `movq` instructions comply with the System V ABI mandate that integer- and pointer-type arguments are, from left to right, passed in the registers RDI, RSI, RDX, RCX, ... The values `30` and `40` fit into the lower 32-bit half of a 64-bit signed integer, which is why a 32-bit copy via `movl` into the lower 32-bit halves EDX and ECX of the 64-bit registers RDX and RCX is sufficient, respectively. For the values `-10` and `-20`, the full 64-bit literal is copied into RDI and RSI, respectively. Integer return arguments are passed in RAX, which is why `addq` adds `50` to that register.

3.2.1.f. Addressing Modes. Listing 3.2 is an example for function arguments of pointer type, and it illustrates that a single x86-64 instruction can load data from memory, process it, and store it, all at once — register names in brackets, like `(%rdi)`, identify the memory location at the address stored in the register. Listing 3.3 illustrates the full complexity of this addressing mode: `-16(%rdi,%rsi,8)` refers to the memory location given by a pointer in RDI, plus 8 bytes times the content of RSI, minus 16 bytes.

The address calculation hardware can be "misused" for regular integer addition, as illustrated in Listing 3.4 where the `leaq` instruction loads the memory address identified by the first operand into the register specified by the second operand.

Listing 3.5 is an example for the related *RIP-relative* addressing mode that enables

Listing 3.3: More complex indirect addressing.

```c
void f(long long* arr,
       unsigned long i){
  arr[i-2] += 100;
}
```

```
f:
  endbr64
  addq  $100, -16(%rdi,%rsi,8)
  ret
```

Listing 3.4: 64-bit integer addition realized by a "load effective address" instruction `leaq` by GCC 11.3.0.

```c
long long f(long long a,
            long long b){
  return a+b;
}
```

```
f:
  endbr64
  leaq  (%rdi,%rsi), %rax
  ret
```

position-independent code, as explained in the following. The `.quad` directive places a four-byte value in the data segment of the object file (as requested by the `.data` directive) and its location is labelled `flipmask`. While it would be possible for the assembler to use a hard-coded address as the first operand of `movq` in the object file, doing so would imply that whenever the code is moved to a different address, these hard-coded addresses would need to be adjusted. Instead, the assembler determines the code position of `flipmask` relative to the `movq` instruction and stores this difference in the machine code. When the `movq` instruction with RIP-relative addressing executes, the difference can be added to the instruction pointer RIP to obtain the address of `flipmask` no matter which memory address the machine code has been loaded to.

3.2.2. Floating-Point Instructions

Regarding floating-point data and arithmetic, x86-64 provides two main sets of registers and instructions, namely, *x87* and *SSE*.

3.2.2.a. X87. The *x87* floating point unit (FPU) is the older system and was originally implemented by a separate Intel 8087 coprocessor sitting besides the 32-bit x86 CPU, and later became part of the main CPU circuitry. It has eight 80-bit registers, which

Listing 3.5: RIP-relative addressing.

```c
unsigned long flipmask = 123456ul;
void f(unsigned long* a){
  *a ^= flipmask;
}
```

```
.text
f:
  endbr64
  movq  flipmask(%rip), %rax
  xorq  %rax, (%rdi)
  ret

.data
flipmask:
  .quad 123456
```

Listing 3.6: Floating point arithmetics via x87 instructions and registers.

```
void f(long double a,
       long double b,
       long double* c){
  *c = a*b;
}
```

```
f:
  endbr64
  fldt   8(%rsp)
  fldt   24(%rsp)
  fmulp %st, %st(1)
  fstpt (%rdi)
  ret
```

can provide a higher floating-point precision than `binary64`. GCC uses x87's 80-bit type to implement the C type `long double`. Listing 3.6 shows that function arguments of this type are passed on the call stack: The first `fldt` instruction loads ten bytes from the memory address in RSP plus 8 bytes, and the second `fldt` loads ten bytes from the address 16 bytes higher (while only 10 bytes are required to store a number in the x87 format, 6 bytes of padding are inserted to align them to addresses divisible by 16 bytes). The eight x87 registers are organized like a stack, meaning that the two floating-point numbers have been loaded into the x87 registers ST(0) and ST(1), and the `fmulp` removes them and pushes their product to ST(0). From there, `fstpt` stores it at the memory address indicated by the pointer-type argument `c`, passed in RDI.

The x87 instruction set even comprises instructions `fsin`, `fcos`, `fptan` to evaluate trigonometric functions, `f2xm1` for exponentiation and `fyl2x` for the logarithm. These instructions could be used to implement the respective functions of the C standard library header `math.h`, unless an implementation in software is preferred. In our work, we did not encounter e. g. `fsin` in actual compiler output or library code. Some of these instructions were found to sometimes having been implemented with bad accuracy.[47,57] Today, software implementations are preferred; see e. g. Listing A.7 in Appendix A.3.

The special-purpose x87 control word register (FCW) carries *rounding control* and *precision control* bits to select a rounding mode and floating-point format, and several *exception mask* bits that control how the processor reacts to floating-point exceptions, e. g. by signaling them through exception flags in the x87 status word register (FSW). FSW also contains the top-of-stack pointer, indicating which of the x87 data registers is ST(0). The tag word register (FTW) reflects which data registers are full (and also distinguishes between valid, zero and special content). Additionally, the x87 non-data processor state comprises pointers to the last non-control x87 instruction and memory operand, and a representation of its opcode.

3.2.2.b. SSE. After x87, further technological progress in x86 floating-point processing went in the direction of parallel *single-instruction-multiple-data* (SIMD) vector operations. The *Streaming SIMD Extensions* (SSE, 1999, later extended multiple times) introduce eight independent 128-bit registers XMM0, . . . , XMM7; the analogous extension to x86-64 introduces sixteen 128-bit registers XMM0, . . . , XMM15. The *Advanced Vector Extension* (AVX, 2008) widens these to 256-bit registers YMM0, . . . YMM7/YMM15. AVX-512 (2013) doubles the number of vector registers, and widens them to 512-bit registers ZMM0, . . . , ZMM31. The media extension control and status register (MXCSR)

Listing 3.7: Floating point arithmetics via SSE/AVX instructions and registers.

```
void f(double a,
       double b,
       double* c){
  *c = a*b;
}
```

```
f:
  endbr64
  vmulsd  %xmm1, %xmm0, %xmm0
  vmovsd  %xmm0, (%rdi)
  ret
```

Listing 3.8: Parallel copy and multiplication of eight `binary64` floating-point numbers using AVX-512 instructions. Obtaining this output required a modified target architecture flag `-march=skylake-avx512` and an additional flag `-mprefer-vector-width=512`.

```
void f(double const* restrict a,
       double const* restrict b,
       double* restrict c){
  for(int i=0; i<8; i++){
    c[i] = a[i]*b[i];
  }
}
```

```
f:
  endbr64
  vmovupd (%rdi), %zmm0
  vmulpd  (%rsi), %zmm0, %zmm0
  vmovupd %zmm0, (%rdx)
  vzeroupper
  ret
```

carries floating-point rounding control bits as well as exception masks and flags.

Listing 3.7 shows that `binary64` arguments are passed in the SSE registers XMM0 and XMM1, and that the AVX instructions `vmulsd` and `vmovsd` are used to multiply and copy them. Listing 3.8, compiled for an AVX512-capable target architecture (`-march=skylake-avx512`) with an additional flag `-mprefer-vector-width=512`, illustrates the AVX-512 instructions `vmovupd` and `vmulpd` copying and multiplying eight `binary64` numbers with a single instruction.

GCC uses the vector registers to copy a single `binary32` from one memory location to another, see Listing 3.9. As this is just a type-agnostic data transfer, loading a 32-bit integer into a general-purpose register and storing it from there, as shown in Listing 3.10, would have the same functional effect.

The vector registers can also be used for integer arithmetic and bitwise logical SIMD operations, as illustrated in Listing 3.11 (compiled with the AVX-512 flags again): The `vmovdqu16` instruction loads the first SIMD vector into ZMM0 and the `vpsubw` instruction performs the subtraction. The `vpbroadcastw` instruction initializes all thirty-two 16-bit components of ZMM1 with the read-only 16-bit integer `1234` identified via RIP-relative addressing, and `vporq` performs the bitwise logical "or". `vzeroupper` zeroes the bits of

Listing 3.9: GCC copied a single `binary32` via the vector registers.

```
void f(float* dest, float* src){
  *dest = *src;
}
```

```
f:
  endbr64
  vmovss  (%rsi), %xmm0
  vmovss  %xmm0, (%rdi)
  ret
```

Listing 3.10: If the source code says so, GCC would copy a single `binary32` via type-agnostic data transfer instructions.

```c
void f(float* dest, float* src){
  *(int*)dest = *(int*)src;
}
```

```
f:
  endbr64
  movl   (%rsi), %eax
  movl   %eax, (%rdi)
  ret
```

Listing 3.11: Parallel copy, subtraction, and bitwise logical "or" operations applied to thirty-two 16-bit signed integers using AVX-512 instructions. Obtaining this output required the same flags as in Listing 3.8.

```c
void f(short const* restrict a,
       short const* restrict b,
       short* restrict c){
  for(int i=0; i<32; i++){
    c[i] = (a[i]-b[i]) | 1234;
  }
}
```

```
f:
  endbr64
  vmovdqu16 (%rdi), %zmm0
  vpbroadcastw  .LC1(%rip), %zmm1
  vpsubw  (%rsi), %zmm0, %zmm0
  vporq %zmm1, %zmm0, %zmm0
  vmovdqu16 %zmm0, (%rdx)
  vzeroupper
  ret
.section  .rodata.cst2, ↩
    "aM",@progbits,2
.LC1:
  .value  1234
```

the YMM and ZMM registers that do not overlap with XMM registers; inserting it seems to have performance reasons.

3.2.3. Further Instructions

3.2.3.a. Conditional Branches. Branch statements and loops in high-level languages can be realized with *conditional branch* instructions. In Listing 3.12, after XMM1 has been zeroed by applying a bitwise logical "exclusive or" to itself, the `vcomisd` instruction compares it with the function argument `a` in XMM0. Specifically, `vcomisd` compares the two lowest-lane `binary64` components and sets specific bits in the special-purpose EFLAGS register. Depending on these bits, `ja` either proceeds to the next instruction that doubles XMM0 by adding it to itself, or jumps to the label `.L8` in order to add 2.0 (which has a `binary64` representation of `4000 0000 0000 0000`, and the integer value of the higher-significant 32 bits of this is 1073741824).

3.2.3.b. Atomic Instructions and Multi-Threading. Modern computers contain multiple processors (also called *cores*) with separate registers, but shared access to memory. In *multi-threaded* programs, several cores intentionally work on shared data. It is often necessary to coordinate access to shared data, in order to avoid *data races* where the result of a computation depends on the otherwise unpredictable order of these accesses.

Listing 3.12: C `if` statement realized with a conditional branch instruction by GCC.

```c
double f(double a){
  if (a<0){
    return 2+a;
  } else {
    return 2*a;
  }
}
```

```
.text
f:
  endbr64
  vxorpd   %xmm1, %xmm1, %xmm1
  vcomisd %xmm0, %xmm1
  ja   .L8
  vaddsd   %xmm0, %xmm0, %xmm0
  ret
.L8:
  vaddsd   .LC1(%rip), %xmm0, %xmm0
  ret

.section   .rodata.cst8,  ↩
   "aM",@progbits,8
.LC1:
  .long 0
  .long 1073741824
```

For example, let us assume that two threads T_1 and T_2 try to increment a shared variable a by 1.0 and 2.0, respectively, by

1. loading a thread-local copy of the shared state a,
2. incrementing their thread-local copy of a, and
3. writing the result back to the shared state a.

Then, it could happen that they both read the original value of a in step (1), later step (3) of T_1 writes $a + 1.0$, and finally step (3) of T_2 overwrites this with $a + 2.0$, so the final result is $a + 2.0$ and the contribution of T_1 is lost. If threads are scheduled differently, the result might as well be $a + 1.0$, $a + 3.0$, or even some other value depending on how exactly everything is implemented.

The x86-64 ISA offers *compare-and-exchange* (also called *compare-and-swap*, CAS) instructions like `cmpxchgq` to facilitate thread-safe updates of a shared state stored at a memory address `addr`. This instruction moves the second operand into RAX, and if this did not change RAX, subsequently moves the first into the second operand. If the second operand refers to a memory address `addr`, the `lock` *prefix* ensures that these operations happen *atomically*; i. e., the processor running the `lock cmpxchgq` instruction has exclusive access to the memory at `addr` while completing the entire instruction.

The `cmpxchgq` instruction also updates the EFLAGS register according to the result of the comparison. Alternatively, whether a CAS was successful or not can be checked by comparing the value of RAX after the CAS instruction with the previously read value of the shared state.

Listing 3.13 shows how GCC uses a CAS instruction. The C code specifies an atomic addition of a shared `binary64` variable at `addr` with `2.0` in the OpenMP multithreading API. The machine code

Listing 3.13: Compare-and-swap instruction used by GCC for an atomic update of a shared state in a multi-threaded OpenMP environment. Compiled with an additional GCC flag `-fopenmp`.

```c
#include <omp.h>

void f(double* addr){
  #pragma omp atomic
  *addr += 2.;
}
```

```asm
.text
f:
  endbr64
  movq   (%rdi), %rdx
  vmovsd   .LC0(%rip), %xmm0
.L2:
  vmovq %rdx, %xmm2
  vaddsd   %xmm2, %xmm0, %xmm1
  movq   %rdx, %rax
  vmovq %xmm1, %rcx
  lock cmpxchgq %rcx, (%rdi)
  jne .L3
  ret
.L3:
  movq   %rax, %rdx
  jmp .L2
.section   .rodata.cst8,   ↩
    "aM",@progbits,8
.LC0:
  .long 0
  .long 1073741824
```

1. loads the present value of the shared state at `addr` (which, as a function argument of pointer type, is passed in RDI), into RDX, and 2.0 into XMM0 (yellow);
2. copies RDX into RAX and computes `*addr` + 2.0 into RCX (red);
3. performs a CAS, writing RCX to `*addr` if `*addr` still matches with the value in RAX (blue); and
4. returns if the CAS succeeded (with `jne` checking the zero flag in FLAGS), and otherwise repeats from step (2) with the new value of `*addr` copied into RDX (gray).

This ensures that no matter how the threads are scheduled, the result will always be (almost) the same. As e. g. floating-point addition is not associative, the binary representation of the result might still depend on the order of operations, but the floating-point value should not change much and this is not considered a data race.

3.2.4. System Calls: Interaction with the Operating System

With conditional and unconditional jumps, data transfer, integer arithmetic, floating-point arithmetic, and logical instructions, we have seen all types of instructions necessary for data processing in a narrow sense. However, we remain to discuss how programs can start threads, allocate memory, and interact with the "outside world" e. g. through standard, file and network I/O. This work focuses on machine code compiled in order to

Listing 3.14: An x86-64 Linux assembly "Hello World" program adapted from Toal[175]. It can be compiled with GCC using the flag `-nostartfiles`.

```
.data
s:
  .string "Hello World!\n"

.text
.globl _start
_start:
  movq $1, %rax         # syscall ID 1 (= sys_write)
  movq $1, %rdi         # output file descriptor stdout (= 1)
  leaq s(%rip), %rsi    # pointer to output data
  movq $13, %rdx        # size of output data (13 bytes)
  syscall               # call write(1,s,13)

  movq $60, %rax        # syscall ID 60 (= sys_exit)
  movq $0, %rdi         # exit status 0 (= OK)
  syscall               # call exit(0)
```

execute in a *userspace* process under the Linux operating system kernel. In most cases, userspace interaction with the "outside world" is, in fact, interaction with the kernel, which in turn accesses the appropriate storage media, network interfaces etc. Processors have a *protection ring* mechanism that allows such accesses only to kernel code running in a high privilege level called *kernel mode* or *ring 0*, while userspace processes run in *user mode* or *ring 3*.

3.2.4.a. System Calls. Userspace programs thus interact with the kernel by means of *system calls* (*syscalls*), which are interrupts or specific instructions that raise the privilege level and jump into code of the kernel. The "calling conventions" of system calls are specific to the kernel; the implementation of the C standard library wraps system calls as normal functions that userspace programs can call in a portable way. But of course, it is possible to make system calls directly from assembly code. As a simple example, the "Hello World" program for x86-64 assembly under Linux in Listing 3.14, adapted from Toal[175], performs two syscalls to Linux on x86-64. In both cases, the `syscall` instruction is used after parameters have been placed in RAX, RDI, RSI, RDX. For the first syscall `write`, RAX holds the syscall ID `1`, RDI contains the file descriptor used for writing, which is `1` for standard output, RSI contains a pointer to the output data, and RDX contains the number of bytes to be written. A second syscall `exit` with syscall ID `60` in RAX and the exit status `0` (success) in RDI terminates the process.

3.2.4.b. Relevance to Real Arithmetic. Floating-point data can thus, apart from being stored in the program's data section (Listing 3.12) or being produced by instructions, enter the memory of a program through syscalls. For instance, libraries implementing the message passing interface (MPI) provide a convenient way to transmit floating-point arrays between processes in a distributed-memory parallel environment.

3.2.5. Shared Libraries

We have already used the term *software library* to refer to collections of pre-defined functions that can be used by different programs. On modern systems, *(dynamic) shared libraries* are located and linked either during program startup, or at run-time. Later in this thesis, we will encounter shared libraries of the two following sorts.

3.2.5.a. The C Standard Library. The C programming language defines a standard library with basic functionality required by many programs, like heap management (e. g. `malloc` and `free`), input/output functions (e. g. `printf`, `fscanf`), copy functions (e. g. `memcpy`, `memmove`), string functions (e. g. `strlen`, `strcat`) and math functions (e. g. `exp`, `log`, `sin`). The GNU implementation of the C standard library is called *glibc*. For historical reasons, math functions are contained in a shared library file `libm.so` separate from `libc.so`.

3.2.5.b. Add-On Mechanisms. Userspace programs can locate and dynamically load shared libraries at run-time. This allows to implement an "add-on mechanism", by which a user can make additional code available to a running program. For instance, Python-C modules are shared objects loaded at run-time by the Python interpreter; we continue on this topic in Section 8.2.

3.2.6. Virtual Memory

All memory addresses handled by userspace programs are *virtual addresses*. For each process, the kernel keeps track of a *page table* that defines the mapping of the virtual address space of the process onto physical memory addresses. This way, different userspace processes can use the same (virtual) address space for their code, data and stack, without interfering with each other. Likewise, it is possible to map different virtual addresses of different processes to a single physical memory page – this allows, e. g., to have a single copy of the non-writable parts (e. g. text sections) of the C standard library (Paragraph 3.2.5.a) and other libraries in physical memory, rather than loading one copy for each process. Listing 3.15 shows how a process can use the `mmap` system call to acquire two different pointers to the same portion of physical memory.

The *memory management unit* (MMU) is a component of the processor hardware that reads the page table and automatically translates virtual into physical addresses for every access to main memory. When the userspace program attempts to access a virtual address that is not mapped, the MMU raises a *page fault* interrupt, giving control to the kernel. "Innocent" page faults like the first write to freshly allocated memory, or access to data that the kernel has swapped to disk and that need to be reloaded, are resolved by the kernel and are functionally transparent to the userspace process. However, if the kernel determines that the userspace program attempted an invalid memory access, it sends a *segmentation violation* signal to the process.

Listing 3.15: Compiled with GCC using glibc on x86-64 Linux, this program acquires two pointers `x`, `y` to the same physical memory address. It therefore prints `42` even though `x` and `y` are different virtual addresses and there is no write to `*y`.

```
#define _GNU_SOURCE
#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(){
  int fd = memfd_create("",0);
  ftruncate(fd,0x1000);
  volatile int* x = (int*)mmap(NULL,0x1000,PROT_READ|↩
     PROT_WRITE,MAP_SHARED,fd,0);
  volatile const int* y = (int*)mmap(NULL,0x1000,PROT_READ,↩
     MAP_PRIVATE,fd,0);
  *x = 42;
  printf("%d\n", *y);
}
```

3.3. "Bit-Tricks": Alternative Ways to Perform Real Arithmetic

In computational science and engineering, the standard way of storing real numbers as digital data is by the three floating-point formats discussed in Section 3.1, and real-arithmetic operations are almost always applied via the respective x87 and/or SSE instructions (Section 3.2.2). In this section, we have a look at alternative ways to perform real arithmetic. Depending on the application scenario, such "bit-tricks" may be employed to improve performance (because they might need less CPU cycles or work on multiple numbers in parallel), because they offer a specific decimal or higher binary accuracy, or just to confuse people.

3.3.1. Alternative Floating-Point Formats

IEEE 754 specifies further floating-point formats besides `binary32` and `binary64`, including decimal formats `decimal32`, `decimal64`, `decimal128`. Their advantage over binary floating-point formats lies in their well-defined decimal accuracy. They are rare in scientific computing software where this kind of accuracy is not required, but according to the literature, business and financial applications frequently rely on decimal formats.[44,185] With the new types `_Decimal32`, `_Decimal64`, `_Decimal128` proposed for the C 23 standard,[98] decimal floating-point arithmetic will become a language feature in C. Several ISAs have added hardware support for decimal formats, with z/Architecture announcing it in 2007,[53,89] PowerPC in 2007 as well,[166] and SPARC64X around 2013.[66,198] On architectures without hardware support like x86-64 and x86, decimal floating-point arithmetic must be implemented in software, using code that operates on decimal floating-point data e. g. in an integer arithmetic and bitwise logical fashion; see Listing 3.16 for an example.

Listing 3.16: GCC uses a software implementation of decimal floating-point arithmetic, calling the function `__bid_adddd3` in `libgcc.so`. The type `_Decimal64` is proposed in the current draft of the C 23 revision, and already supported as a GNU extension.

```
_Decimal64 f(_Decimal64 a,
             _Decimal64 b){
  return a + b;
}
```

```
f:
  endbr64
  subq   $8, %rsp
  call   __bid_adddd3@PLT
  addq   $8, %rsp
  ret
```

Of course, such an "emulated FPU" can always be used, even for `binary32` and `binary64` data, instead of the suitable floating-point instructions.

Related to that, the GNU Multiple Precision Floating-Point Reliable Library (MPFR)[61] defines many non-standard floating-point formats, offering e. g. arbitrarily high floating-point precision. To make them usable, the MPFR implements the corresponding elementary real-arithmetic operations and math functions. When the MPFR or an emulated FPU is employed in the primal program, what runs on the CPU are these implementations. Their underlying real-arithmetic meaning is by no means obvious on the machine code level.

3.3.2. Manipulation of the Sign Bit

According to equation (3.1), the absolute value of a `binary64` (and also `binary32` or x87 extended precision) floating-point number can be formed by simply setting the sign bit to `0`. This can be achieved by applying a bitwise logical "and" operation to the floating-point representation and a 64-bit (or 32-bit or 80-bit) constant `0b01...11`. Listing 3.17 demonstrates that GCC routinely uses this "trick" to compute absolute values without using any floating-point instruction. Note that the 64-bit constant `0b01...1` is composed from a 32-bit `0b1...1` in the lower half and a 32-bit `0b01...1` in the upper half, and zero-padded into a 128-bit operand.

Similarly, the negative value of a floating-point number can be computed by a bitwise logical "exclusive or" with `0b10...0`, and the negative absolute value can be computed by a bitwise logical "or" with `0b10...0`.

When both operands of an "exclusive or" or "or" are `0b10...0`, it is not clear which operand represents the real number (if any). As a floating-point operand, `0b10...0` represents -0.0 according to equation (3.1), so computing the negative of -0.0 can lead to such an ambiguous use of the "exclusive or" instruction. With the "and" operation, there is no ambiguity because if `0b01...11` were read as a `binary64` or `binary32`, it would represent NaN.

Listing 3.17: GCC 11.2.0 may use a 128-bit logical “and” to set the sign bit of a `binary64` to zero, in order to compute the absolute value. We have inserted comments into the assembly code to indicate the bitwise representations of the four 32-bit constants.

```
#include <math.h>
double f(double x) {
  return fabs(x);
}
```

```
f:
  endbr64
  andpd .LC0(%rip), %xmm0
  ret
; ...
.LC0:
  .long -1          ; 0b11..11
  .long 2147483647  ; 0b01..11
  .long 0           ; 0b00..00
  .long 0           ; 0b00..00
```

Listing 3.18: Clang 14.0.0 may use logical operations in a masking pattern to select one of two floating-point numbers.

```
double f(double a){
  if (a<0){
    return 2+a;
  } else {
    return 2*a;
  }
}
```

```
.section  .rodata.cst8, ↩
   "aM",@progbits,8
.LCPI0_0:
  .quad 0x4000000000000000
.text
f:
  xorpd %xmm1, %xmm1
  movapd  %xmm0, %xmm2
  cmpltsd %xmm1, %xmm2
  movsd .LCPI0_0(%rip), %xmm1
  andpd %xmm2, %xmm1
  andnpd  %xmm0, %xmm2
  orpd  %xmm1, %xmm2
  addsd %xmm2, %xmm0
  retq
```

3.3.3. Masking

When we compile Listing 3.12 with Clang for the target `-march=nehalem`, we obtain the assembly code in Listing 3.18. Note that it uses *masking* instead of a conditional jump: The `cmpltsd` instruction compares XMM1 (zeroed by the `xorpd` instruction and thus representing `+0.0`) and XMM2 (initialized with `a` which has been passed in XMM0), and sets XMM2 to either `0xff...ff` if $a < 0$, or `0x00...00` otherwise. The subsequent “and”, “and-not” and “or” instructions use this *mask* in a masking expression

$$\texttt{(mask and value\_if\_true) or ((not mask) and value\_if\_false)} \tag{3.3}$$

to select either the constant 2.0 copied from `.LCPI0_0`, or the value a copied from XMM0, respectively, into XMM2. From there it is added into XMM0, so the function returns $\mathsf{a} + 2$ or $\mathsf{a} + \mathsf{a} = 2\mathsf{a}$.

Listing 3.19: Possible implementation of the `frexp` function of `math.h` for normal numbers, applying bitwise logical and shift instructions to the `binary64` argument.

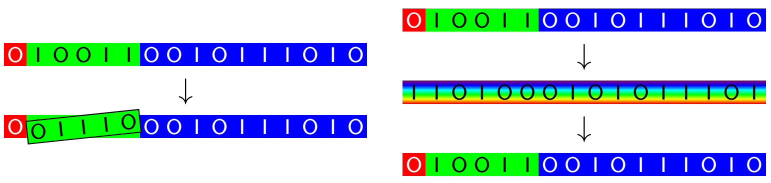
```
typedef unsigned long long ull;
ull const E_mask
  = (0b11111111111ul << 52);
ull const E_value
  = (0b01111111110ul << 52);

double f(double arg, int* exp){
  ull arg_ul = *(ull*)&arg;
  *exp = (arg_ul & E_mask) >> 52;
  *exp -= 1022;
  ull result
    = (arg_ul & ~E_mask) | E_value;
  return *(double*)&result;
}
```

```
f:
  endbr64
  vmovq %xmm0, %rdx
  shrq  $52, %rdx
  andl  $2047, %edx
  subl  $1022, %edx
  movl  %edx, (%rdi)
  vmovq %xmm0, %rax
  movabsq $-9218868437227405313, ↩
     %rdx
  andq  %rdx, %rax
  movabsq $4602678819172646912, ↩
     %rdx
  orq %rdx, %rax
  vmovq %rax, %xmm0
  ret
```

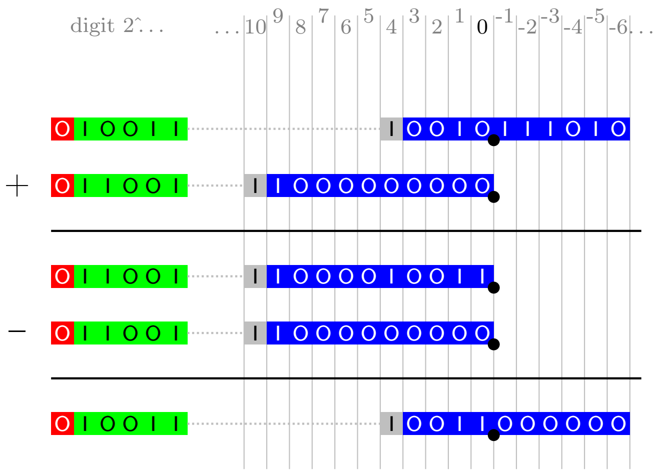
3.3.4. Manipulation of the Exponent Bits

3.3.4.a. Overwriting. The masking expression (3.3) can also be used to overwrite the exponent bit field, e. g. in order to implement the `frexp` function from the C header `math.h`. Except for corner cases, `frexp` takes a `binary64` `arg` and a pointer `exp` to an integer as arguments, assigns a signed integer $-k$ to `*exp` such that $b = 2^k \cdot \texttt{arg}$ satisfies $\frac{1}{2} \leq |b| < 1$, and returns b. This real-arithmetic operation can be implemented as in Listing 3.19 without any floating-point operations, extracting k from the eleven exponent bits of the `binary64`, and overwriting them with `0b01111111110`, which lets $E = -1$ in (3.1). The code in Listing 3.19 has been inspired from code of the VDT math library[154] used in the Geant4 software; see Appendix A.1 for details. The idea is illustrated in Figure 3.2a.

3.3.4.b. Integer Addition. The “reverse” operation `ldexp`, which multiplies a floating-point argument with a power 2^k for a given integer argument k, can likewise be implemented by an integer addition of k to the exponent bits. Listing 3.20 illustrates this with the `binary32` variant `ldexpf`, for which the integer arguments must be shifted by 23 bits to the left so it becomes aligned with the eleven exponent bits. Note that the code does not work correctly in corner cases like subnormal numbers. We observed this kind of bit-trick in a SIMD version of `ldexpf` used by NumPy[78] v1.19.5, with the code in Listing A.5 in Appendix A.2. NumPy calls this function in the implementation of the exponential function for a `binary32` SIMD vector in Listing A.3, where initially a multiple $k \cdot \ln 2$ is subtracted from the argument to map it into a suitable range for an approximating rational function, and the result is scaled by 2^k to account for this range reduction. See Appendix A.2 for details.



(a) Overwriting of exponent bits, Paragraph 3.3.4.a. (b) Complicated binary identity, Section 3.3.6.



(c) Using floating-point errors for rounding, Section 3.3.5.

Figure 3.2: Illustrations of a few bit-tricks from Section 3.3.

Listing 3.20: Possible implementation of the `ldexpf` function of `math.h` for normal numbers, applying integer addition and shift instructions to the `binary32` argument. This is just a simple sketch; the code does not work in some corner cases.

```
unsigned int float_to_int(float x){
  return *(unsigned int*)&x;
}
float int_to_float(unsigned int x){
  return *(float*)&x;
}
float f(float arg, int exp){
  return int_to_float(
    float_to_int(arg) + (exp<<23)
  );
}
```

```
f:
  endbr64
  sall   $23, %edi
  vmovd  %xmm0, %eax
  addl   %edi, %eax
  vmovd  %eax, %xmm0
  ret
```

3.3.5. Exploiting Floating-Point Inaccuracies for Rounding

For a `binary64` number x in the range between $2^{52} \leq |x| < 2^{53}$, the exponent E in formula (3.1) is 52. Thus, the least-significant bit b_0 of the significand controls the binary digit $2^{E-52} = 1$. Therefore within the above range for x, the real numbers representable in the `binary64` floating-point format are precisely the integral numbers. This fact can be exploited to implement a rounding operation for a `binary64` y with $|y| < 2^{51}$, as follows.

In a first step, $T = 1.5 \cdot 2^{52}$ is added to y. As $2^{52} < |T + y| < 2^{53}$, storing the sum as a `binary64` rounds it to an integral number, obeying the present rounding mode (Paragraph 3.1.d). In a second step, T is subtracted again from the result of $(T + x)$. This does not introduce any more floating-point errors, so we end up with the value of y rounded to an integral number. Figure 3.2c illustrates this bit-trick using the `binary16` format.

Unlike the previously presented bit-tricks, this one uses floating-point instructions. However, the intended arithmetic effect comes from inaccuracies of the specific binary representation used to store $(T+x)$, rather than the semantic of addition and subtraction in terms of infinitely precise arithmetic.

Depending on the possible range of x, other constants between 2^{52} and 2^{53} may be used for T instead of $1.5 \cdot 2^{52}$, and the subtraction step may be split into multiple operations scattered around in the machine code. It would therefore be extremely hard to spot this kind of bit-trick in all of its possible variations. And even if it were detected, it would remain unclear whether the intention behind the code is to perform a rounding operation, as it might as well just be a badly conditioned coincidental pair of an addition and subtraction that together incur a large floating-point error.

The glibc math library makes frequent use of this kind of bit-trick, e. g. for range reduction or to compute an index into a lookup table, see Appendix A.3. Also, GCC may use it to implement the math function `rint`, as we show in Listing 3.21. Note that in this example, we specified the target architecture `x86-64` and not `skylake`, where GCC uses a dedicated `vroundsd` instruction instead, and that also the `-O3` optimization level is important as unoptimized builds (`-O0`) use a call to the math library function `rint` as expected.

Finally, the previous NumPy `expf` example in Listing A.3 contains the rounding bit-trick as well.

3.3.6. Instruction Sequences Composing a Binary Identity

For the purpose of encoding, encryption or compression, computer programs may subject binary data to sequences of instructions that temporarily modify it, but restore the original values in the end. For example,

- Listing 2.2 converts a floating-point number to a string representation and back;
- Listing 3.22 shows a binary identity employed by GCC to copy data for an OpenMP atomic update on x86; and

Listing 3.21: GCC 11.4.0 with `-O3` and `-march=x86-64` uses floating-point inaccuracies to implement calls to the rounding function `rint`, as outlined in Section 3.3.5. `.LC0` is a `binary64` representation of 2^{52} and `.LC1` is `0x7f...ff`. The `andpd` instruction calculates the absolute value of the function argument (Section 3.3.2). `jbe` jumps right to the end of the function if the absolute value compares larger or equal to 2^{52} with `ucomisd`, because then no rounding is necessary. Otherwise, 2^{52} in XMM3 is first added (`addsd`) and then subtracted (`subsd`); the bitwise logical `andnpd` and `orpd` instructions are in place to preserve the sign.

```c
#include <math.h>

double f(double x){
  return rint(x);
}
```

```
f:
  endbr64
  movsd .LC1(%rip), %xmm2
  movsd .LC0(%rip), %xmm3
  movapd  %xmm0, %xmm1
  andpd %xmm2, %xmm1
  ucomisd %xmm1, %xmm3
  jbe .L2
  addsd %xmm3, %xmm1
  andnpd  %xmm0, %xmm2
  subsd %xmm3, %xmm1
  orpd  %xmm2, %xmm1
  movapd  %xmm1, %xmm0
.L2:
  ret
```

- a hard-to-spot binary identity can also appear when different virtual addresses point to the same physical memory address, as in Listing 3.15.

When the binary data comprises representations of real numbers, correct AD handling requires to recognize the identity in order to keep their AD information.

3.3.7. How to Avoid Creating Bit-Tricks

As outlined in this section, bit-tricks can perform real arithmetic in ways that are very difficult to recognize. We should therefore remark on practical options to avoid inserting them into a compiled program.

We may anticipate from Section 8.1 that while popular compilers like GCC or Clang do use sign bit manipulations (Section 3.3.2) and masking expressions selecting entire numbers (Section 3.3.3), it is very rare that they use other bit-tricks by themselves; though that happened as well in Listing 3.21, where it could be avoided with `-O0`, and in Listing 3.22, where it can be avoided e. g. by not using OpenMP. Most of our examples in this section, however, arise from high-performance math library source code that explicitly operates on floating-point data in a binary way. Such code patterns should be replaced by the appropriate floating-point operations and when they occur in library functions, there are at least three options:

- Maybe it is possible to configure the library in a way that turns off these kinds of

Listing 3.22: GCC 11.2 on `godbolt.org`[72] with flags `-fopenmp` `-m32` `-O3` `-march=pentium` compiling Listing 3.13, using a bit-trick to copy a `binary64`. The argument `addr` is passed at `4(%esp)` according to the x86 Linux calling conventions[118]; after the first three instructions have decreased ESP by $4 + 4 + 28 = 36$ bytes, `addr` is copied from `40(%esp)` to ESI. The `binary64` number `*addr` is then copied to `%esp` with the combination of a `fildq` and a `fistpq` instruction, highlighted in blue. The `fildq` instruction loads the 64-bit signed integer operand as an 80-bit floating-point number into ST(0), and `fistpq` performs the reverse store operation. Chaining `fildq` and `fistpq` in this way has the effect of `movq (%esi) (%esp)` (if this were a well-formed instruction), because the 80-bit type with 64 significand bits plus a sign bit can accurately represent 64-bit integers.
Additionally, the check whether the CAS was successful is highlighted in yellow. To this end, the value of `*addr` just before the CAS (stored in EAX and EDX after the CAS) is compared to the local copy that was incremented by 2.0 (at the top of the stack, `(%esp)`), by means of bitwise logical "exclusive-or" operations and an "or" operation. These bitwise operations affect the zero flag, which controls the conditional jump instruction `jne`.

```
f:
        pushl   %esi
        pushl   %ebx
        subl    $28, %esp
        movl    40(%esp), %esi
        fildq   (%esi)
        fistpq  (%esp)
.L2:
        fldl    (%esp)
        fadds   .LC0
        movl    (%esp), %eax
        movl    4(%esp), %edx
        fstpl   8(%esp)
        movl    8(%esp), %ebx
        movl    12(%esp), %ecx
        lock cmpxchg8b  (%esi)
        movl    (%esp), %ebx
        movl    4(%esp), %ecx
        xorl    %eax, %ebx
        xorl    %edx, %ecx
        orl     %ebx, %ecx
        jne     .L3
        addl    $28, %esp
        popl    %ebx
        popl    %esi
        ret
.L3:
        movl    %eax, (%esp)
        movl    %edx, 4(%esp)
        jmp     .L2
.LC0:
        .long   1073741824
```

optimizations. This option works, e. g., for the bit-tricks in NumPy's exponential function (see Paragraph A.2.d).

- If the source code of the library is accessible and it is not too much work, one might also modify the source code to eliminate bit-tricks; this is what we did in Geant4's `G4Log.hh` header (Paragraph 10.3.e).
- One can try to recognize calls to library functions by their symbol names, and reroute them to wrapper functions. We use this solution to work around bit-tricks in the C math library (Section 5.5).
- Otherwise, one should try to find alternatives to the library.

3.4. Summary

Automatic differentiation, as outlined in Chapter 2, is based on the full knowledge of the real-arithmetic expression tree evaluated by the primal program. It is therefore critical to recognize all the real-arithmetic calculations performed by the primal program. There are three widespread floating-point formats, and the x86-64 and x86 ISAs have specific instruction subsets for floating-point arithmetic. Besides floating-point instructions, type-agnostic copy operations are relevant for AD because they can copy floating-point data. In Chapter 4, we describe the practical aspects of recognizing (the execution of) these instructions in the primal program.

We have outlined several further ways how real arithmetic can be "hidden" in a portion of machine code, e. g. by bitwise manipulation of the sign or exponent bits, by software emulation of a floating-point unit, or by exploiting the limited floating-point accuracy for rounding operations. We lack a systematic understanding of all of the possible bit-tricks. This would be a fundamental obstacle if we sought to create a truly universal AD tool that could even handle hand-written assembly code of a determined counterexample-maker.

However, we take a more practical perspective. Derivgrind currently supports manipulations of the sign bit (Section 3.3.2), masking expressions for entire floating-point representations (Section 3.3.3), and whatever happens inside the C math library functions. Apart from this, *we assume that the primal program does not use any other bit-trick.* For users, this mainly means that in the source code of the primal program including libraries, real-arithmetic operations should always be performed by the corresponding language constructs. Next to this, disabling OpenMP is a good idea and in rare cases, unoptimized builds might be required. Thus, we believe that our assumption is only a minor limitation regarding the productive use of AD for compiled programs.

4. Dynamic Binary Instrumentation

4.1. Dynamic Binary Instrumentation Frameworks and Tools

As illustrated in Section 3.2, compiled computer programs on x86-64 Linux systems are *very* complex “data structures”. The goal of this work is to find out about the real-arithmetic operations performed by a program, and to insert the appropriate AD logic; specifically, either forward-mode logic propagating dot values (Section 2.3), index-handling and tape-recording logic (Section 2.4.5), or logic implementing a bit-trick detection heuristic.

4.1.1. Interacting with Machine Code

Possible ways to insert the AD logic into the binary machine code of the primal program may comprise

- *static binary instrumentation*, operating on the machine code of the build artifacts (executable and user library files) without running the code, similar to the forward-mode prototype adac by Gendler et al.[69] (operating on the assembly level); or
- *dynamic binary instrumentation* (DBI), operating on the machine code of the program after it has been loaded into memory, and shortly before it executes.[135]
- Alternatively, a “differentiated CPU” or “differentiated CPU emulator” might run machine code and perform AD logic alongside. Schoder[164] used an FPGA to run a RISC-V core with additional circuitry performing forward-mode AD logic. Related but concerned with a different kind of extra logic, Jurczyk et al.[103,104] used the instrumentation API of the *Bochs* emulator to detect kernel race conditions and memory disclosures.

The operator-overloading, source-transformation and compiler-based approaches (Sections 2.5 to 2.7) taken by source-code based AD tools, when applied to compiled languages, have a similar effect as the static instrumentation approach. When programs grow or change at run-time by dynamic loading of shared libraries, self-modification, or the generation of new machine code, the static approach has no chance to adapt. Conversely, dynamic instrumentation can access the entire userspace machine code of a process, including dynamically linked or loaded user and system libraries and on-the-fly-generated code, at every point of time while it executes; only kernel code is typically not accessible. This amount of universality is suitable for almost all purposes, and it is the approach we take in the following. A differentiated CPU or emulator would be even more universal because it can see all instructions performed on the CPU, by all processes and by the kernel.

4.1.2. Dynamic Binary Instrumentation

This thesis is concerned with the DBI approach to insert AD logic. Its implementation from scratch would, at least, involve the following difficult and error-prone tasks:

- Handling the ELF files of the primal program executable and all dynamically linked or loaded libraries; the dynamic loader of the system might potentially be used for this, but this is not straightforward; also, we might want to have access to information present in the executable and library files but not in the loaded machine code, such as debug symbols;
- repeatedly disassembling portions of the binary machine code when we know at which bytes instructions start;
- adding AD logic, making sure that its use of registers and memory does not interfere with the primal instructions;
- as this stretches the code, adapting any references like RIP-relative addresses;
- making sure to never lose control about which instructions will execute – e. g., jump instructions leaving the instrumented code portion must be redirected back to the DBI infrastructure, which has to instrument the code at the jump target first;
- running this instrumented code; and,
- in parallel, providing suitable interfaces for the user of the tool.

Fortunately, a lot of this work has already been done by authors of *DBI frameworks* like Valgrind[139,168], PIN[120] and DynamoRIO[37,38]. These frameworks take over many of the above tasks, so the author of a DBI tool can focus on the core logic and functionalities of their tool. All these frameworks have in common that they provide a "base" or "core" system that can load and run userspace machine code of a *client program* without ever losing control, and they offer an API for "tools" to manipulate the code, intercept function calls, register callbacks for various types of events, and/or interact with the user. Many of such DBI tools have been created and reported in the literature; we give an overview on the various application domains of DBI in the next section 4.1.3.

4.1.3. Overview on Dynamic Binary Instrumentation Tools

4.1.3.a. Memory Error Detectors. Valgrind's default tool *Memcheck*[167] and DynamoRIO's *Dr. Memory*[36] are *memory checkers.* For each byte of virtual memory, they monitor whether it can be legally accessed by the client program (*valid-address bit*). For each bit of data in virtual memory and registers, they monitor whether it has been properly initialized (in the sense that it does not depend on the content of uninitialized memory; *valid-value bit*). The instrumentation inserted by these tools updates the valid-address and valid-value metadata, and reports warnings if the client program performs invalid memory accesses or if the outside behavior of the program depends on the content of uninitialized memory. Memory checkers need to intercept and instrument calls to memory allocation routines, and can report related errors like double frees or memory leaks.

4.1.3.b. Other Memory-Related Tools. *Cudagrind*[21] extends Valgrind's Memcheck tool with wrapper functions to include data transfers with accelerator devices like GPUs via CUDA drivers into memory error checking. *Annelid*[136] tracks memory ranges for pointers, in order to diagnose memory accesses outside proper bounds.

Taint analysis tools like *TaintCheck*[140], *Flayer*[52] and *Taintgrind*[108] implemented in Valgrind, and *TaintTrace*[41] implemented in DynamoRIO, use metadata with a similar behaviour as Memcheck's valid-value bits to monitor whether data depends on unsanitized input from an untrusted user. A warning is triggered when such data controls "dangerous" operations, e. g. if jumps are performed using tainted target addresses.

4.1.3.c. Thread Error Detectors. The Valgrind tools *DRD* and *Helgrind* can be used to debug threading errors like data races and the possibility of deadlocks. To this end, they record the order of memory accesses and intercept calls to popular threading libraries, mainly the POSIX threading (pthreads) primitives.

4.1.3.d. Profilers. Heap profilers like Valgrind's *Massif* tool, and cache and branch prediction profilers like Valgrind's *Cachegrind* and *Callgrind*[188], can be used to analyze the performance of the client program.

4.1.3.e. Miscellaneous Tools.

- Valgrind's *Redux* tracing tool[137] records the full computational history of a client program by storing every value-producing operation that it performs.
- *FITIn*[80], implemented in Valgrind, injects bit-errors to evaluate software-implemented hardware fault tolerance mechanisms.
- The *pmemcheck*[106] and *Persistent Memory Analysis Tool (PMAT)*[101], both implemented in Valgrind, check the crash consistency of persistent memory applications.

4.1.3.f. Tools Related to Real Arithmetic. A few Valgrind tools put a particular emphasis on instrumenting floating-point instructions, for purposes associated to the mathematical field of numerical analysis:

- *FpDebug*[24] keeps track of the value of floating-point numbers in an arbitrary-precision floating-point type of the MPFR library[61], in order to identify floating-point accuracy problems. To our understanding, the present version instruments only a small subset of the x86-64 floating-point instructions, does not intercept calls to the C math library, and is not aware of any bit-tricks listed in Section 3.3.

- *Herbgrind*[160] shadows floating-point numbers with arbitrary-precision MPFR numbers as well. Additionally, to facilitate the identification of root causes of numerical errors, it applies a taint analysis to track error propagation, and attempts to record symbolic arithmetic expressions of parts of the client program. Herbgrind intercepts calls to the C math library and recognizes the bit-tricks related to the sign bit (Section 3.3.2).
- *Verrou*[58,67,68,74] lets the user choose an "real-arithmetic back-end" that replaces the original floating-point operations performed on the x87 or SSE floating-point unit, and functions in the C math library. The following back-ends are available:
  - *Deterministic Rounding*, fixing one of the IEEE-754 rounding modes (Paragraph 3.1.d), or a "round to farthest" mode maximizing the round-off error;
  - *Random Rounding*, i. e. deciding stochastically whether to round towards positive or negative, either equiprobably or in such a way that the expected outcome matches the value before rounding;
  - *Reduced Precision*, applying `binary32` rounding to `binary64` numbers.

4.2. Instrumentation with Valgrind and VEX

We have implemented the machine-code-based AD tool *Derivgrind* using the Valgrind DBI framework. In this section, we give an overview on the relevant Valgrind internals.

4.2.1. Instrumentation Workflow

After invoking

valgrind --tool=⟨*tool*⟩ ⟨*tool options*⟩ ⟨*client executable*⟩ ⟨*arguments for client*⟩ (4.1)

the Valgrind core starts to read portions of the machine code of the *client program*, which consists of the client executable as well as dynamically linked or loaded libraries, and translates them into an object-oriented intermediate representation called *VEX*. The selected Valgrind tool registers a function with the signature

```
IRSB* tool_instrument ( VgCallbackClosure* closure,
  IRSB* sb_in,
  const VexGuestLayout* layout, const VexGuestExtents* vge,
  const VexArchInfo* archinfo_host,
  IRType gWordTy, IRType hWordTy )
```
(4.2)

During what one might call "instrumentation phases", the Valgrind core invokes this function for portions of VEX code called *superblocks*. Superblocks are represented as instances of the C type `IRSB`. The instrumentation function receives the original superblock through the argument `sb_in`, and returns a pointer to the instrumented VEX

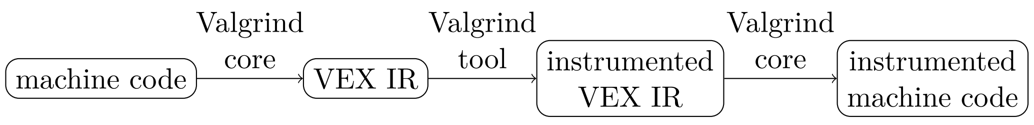


Figure 4.1: The Valgrind instrumentation workflow.

Listing 4.1: Example of the textual representation of a VEX IR superblock, from the documentation of Valgrind[168]. Every line represents a statement. We have emphasized parameters and VEX expressions.

```
------ IMark(0x24F275, 7, 0) ------
t3 = GET:I32(0)               # get %eax, a 32-bit integer
t2 = GET:I32(12)              # get %ebx, a 32-bit integer
t1 = Add32(t3,t2)             # addl
PUT(0) = t1                   # put %eax
```

superblock, constructed using an object-oriented API. For technical reasons, Valgrind tools must not use the C standard library; Valgrind offers replacements for a substantial subset of C standard library functions.[168]

During what one might call “run phases”, the instrumented VEX code is executed. From the perspective of the tool, it suffices to imagine that Valgrind simulates a *synthetic CPU* that interprets the instrumented VEX code, although in fact, Valgrind translates it into machine code and jumps to the generated code.

Valgrind switches between instrumentation and run phases in a *just-in-time* fashion. The instrumented superblocks are cached, so ideally, the run-time of the instrumented program is dominated by the execution of instrumented code during run phases and not by the translation and instrumentation procedures during instrumentation phases.

In the following sections, we have a closer look at the VEX language and execution model. More details on VEX can, of course, be found in the documentation of Valgrind. Though VEX is a binary format with an object-oriented C API, we use Valgrind’s textual representation to display it in this thesis. Listing 4.1 shows a VEX superblock that could correspond to a `addl %eax, %ebx` instruction; this example was taken from the Valgrind documentation[168].

4.2.2. Synthetic CPU

As VEX IR strives for hardware independence, its execution model is built around a synthetic CPU. The synthetic CPU has access to memory using the same virtual addresses as the client program. Registers of the synthetic CPU are specified by a byte offset into a separate block of memory called the *guest state*; a selection of mappings from x86-64 registers to byte offsets is shown in Table 4.1. In addition, VEX IR provides *temporaries* to store intermediate values for the scope of the current superblock. Temporaries are specified by a non-negative integer index and can be assigned only once per superblock.

Table 4.1: Mapping from x86-64 registers to byte offsets in the VEX guest state. The eight 80-bit registers ST(0), ..., ST(7) are represented in the guest state by eight 64-bit segments at the offsets 776, 784, ..., 832, using circular indexing. The total size of the x86-64 guest state is 928 bytes for Valgrind version 3.18.1.

RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI	RIP	YMM0	YMM1
16	24	32	40	48	56	64	72	184	224	256

4.2.3. VEX Statements

Listing 4.1 is the textual representation of a VEX superblock (`IRSB`) from the Valgrind documentation. Each superblocks contains a list of VEX *statements* (`IRStmt`), represented by separate lines in the textual representation. A statement represents an action with side effects, such as writing to a memory address, a register or a temporary. For example, the three lines of the form `t`⟨*index*⟩ `=` ⟨...⟩ in Listing 4.1 represent writes to temporaries. Further kinds of statements include jumps as well as CAS statements and *dirty calls* (see below); Table 4.2 lists most types of VEX statements. In the end of each superblock, the address of the next instruction is stored separately, and indicated in the textual representation with a register write to RIP followed by the jump type.

4.2.3.a. CAS statements. As outlined in Paragraph 3.2.3.b, CAS instructions on x86-64 perform a comparison and affect the EFLAGS register, and perform copy operations conditional to the outcome of the comparison, in an atomic step. Valgrind uses the CAS statement in VEX to implement the core of this mechanism, but the full VEX equivalent to a `lock cmpxchgq` instruction is composed of more than that, as shown in Listing 4.2. In particular, the effect on the EFLAGS register is modelled in VEX by writing the operands to a specific region of the guest state (at byte offsets from 144 inclusive to 176 exclusive). To access flags, VEX code performs a CCall (explained later in Table 4.4) to `amd64g_calculate_condition`.

4.2.3.b. Dirty calls. Dirty call statements allow to run arbitrary code from VEX. Valgrind uses dirty calls to represent instructions in the client program that have no equivalent in VEX, such as the `rdtsc` instruction on x86-64 that reads the time-stamp counter into processor registers, or certain instructions related to the 80-bit x87 floating-point format (Paragraph 4.2.5.a). A dirty call statement `*stmt` can be identified by its name string

`stmt->Ist.Dirty.details->cee->name`. (4.3)

Table 4.3 lists some dirty calls that we have encountered on x86 and x86-64 client programs.

Table 4.2: Overview on types of VEX statements (Section 4.2.3). Some statement types depend on $\boxed{\textit{VEX expressions}}$ (Section 4.2.4) and/or $\langle\textit{parameters}\rangle$.

Textual representation	Effect
`t`$\langle i\rangle$ `=` $\boxed{x}$	Store x in a temporary with index i.
`PUT(`$\langle j\rangle$`) =` $\boxed{x}$	Store x in the register with byte offset j in the guest state.
`PUTI(`$\langle\textit{base}\rangle$`:`$\langle n\rangle$`x`$\langle\textit{type}\rangle$`)[`$\boxed{\textit{ix}}$`,` $\langle\textit{bias}\rangle$`] =` $\boxed{x}$	Store x in the register at $\textit{base}+((\textit{ix}+\textit{bias}) \bmod n)\cdot s$, where s is the size of *type* (circular indexing).
`STle(`$\boxed{\textit{address}}$`) =` $\boxed{x}$	Store x in memory at the address *addr*, using the little-endian storage order.
`if (`$\boxed{\textit{guard}}$`) STle(`$\boxed{\textit{address}}$`) =` x	Store x in memory, if a boolean expression *guard* evaluates to true.
`t`$\langle i\rangle$ `= if (`$\boxed{\textit{guard}}$`)` $\langle\textit{conv}\rangle$`(LDle(`$\boxed{\textit{address}}$`)) else` $\boxed{\textit{alt}}$	If *guard* evaluates to true, assign `t`$\langle i\rangle$ with data from a memory *address* after some possible widening according to *conv*. Otherwise, assign `t`$\langle i\rangle$ with *alt*.
`t`$\langle\textit{old}\rangle$ `= CASle(`$\boxed{\textit{addr}}$ `::` $\boxed{\textit{expd}}$ `->` $\boxed{\textit{new}}$`)`	Compare-and-swap, loading *addr* into temporary `t`$\langle\textit{old}\rangle$, and replacing data at *addr* by *new* if it matches *expd*; see Paragraph 4.2.3.a.

`t`$\langle i\rangle$ `= DIRTY` $\boxed{\textit{guard}}$ `:::` $\langle\textit{name}\rangle$`{`$\langle\textit{addr}\rangle$`}(`$\boxed{\textit{arg1}}$`,` $\boxed{\textit{arg2}}$`, ...)`
 or
 `DIRTY` $\boxed{\textit{guard}}$ `:::` $\langle\textit{name}\rangle$`{`$\langle\textit{addr}\rangle$`}(`$\boxed{\textit{arg1}}$`,` $\boxed{\textit{arg2}}$`, ...)`
	Dirty call to a function *name* in the Valgrind or tool code at *addr*, with potential side effects. Table 4.3 lists a few examples. Arguments are obtained by evaluating *arg1*, *arg2* etc., and a return value can be stored in `t`$\langle i\rangle$. Beyond what is shown here, the VEX statement may specify side effects, certain calling convention details, and details useful for the Valgrind Memcheck tool.

`------ IMark(...) ------`	Meta information.
`if (`$\boxed{\textit{guard}}$`) goto {`$\langle\textit{jump kind}\rangle$`}` $\boxed{\langle\textit{target}\rangle}$	Conditional jump; represents e. g. call/return instructions, system calls, or instructions triggering the execution of an interrupt service routine.

Listing 4.2: Part of the Valgrind output for an OpenMP-parallel program with the Valgrind flags listed in Section 4.2.6. It shows how a `lock cmpxchgq` instruction (Paragraph 3.2.3.b, Listing 3.13) is translated to VEX IR.

```
0x496770C:  lock cmpxchgq %r9,(%rdx)

            ------ IMark(0x496770C, 5, 0) ------
            t7 = GET:I64(32)
            t2 = GET:I64(88)
            t1 = GET:I64(16)
            t3 = CASle(t7::t1->t2)
            PUT(144) = 0x8:I64
            PUT(152) = t1
            PUT(160) = t3
            PUT(168) = 0x0:I64
            t6 = 64to1(amd64g_calculate_condition[mcx=0x13↩
                ]{0x5814c4a0}(0x4:I64,GET:I64(144),GET:I64↩
                (152),GET:I64(160),GET:I64(168)):I64)
            t5 = ITE(t6,t1,t3)
            PUT(16) = t5
            PUT(184) = 0x4967711:I64
```

Table 4.3: Some dirty calls encountered by us when applying Valgrind to x86 and x86-64 client programs.

Name (4.3)	Actions
With potential real-arithmetic effect:	
`x86g_dirtyhelper_loadF80le`, `amd64g_dirtyhelper_loadF80le`	Loads 80-bit extended precision double from memory, converts it to `binary64` and stores it in the return temporary. Used to represent the `fldt` instruction (see Paragraph 3.2.2.a).
`x86g_dirtyhelper_storeF80le`, `amd64g_dirtyhelper_storeF80le`	Converts `binary64` to 80-bit extended precision double and stores it in memory. Used to represent the `fstpt` instruction (see Paragraph 3.2.2.a).
No real-arithmetic significance expected:	
`amd64g_dirtyhelper_FLDENV`, `amd64g_dirtyhelper_FSTENV`	Loads or stores x87 non-data processor state (see Paragraph 3.2.2.a).
`amd64g_dirtyhelper_PCMPxSTRx`	Used to represent SSE 4.2 string instructions.
`x86g_dirtyhelper_CPUID_sse0`, `x86g_dirtyhelper_CPUID_sse3`, `amd64g_dirtyhelper_CPUID_avx2`, etc.	Used to represent `cpuid` instruction, places information on the CPU manufacturer and capabilities in certain registers.
`x86g_dirtyhelper_RDTSC`, `amd64g_dirtyhelper_RDTSC`	Stores time-stamp counter in the return temporary. Used to represent the `rdtsc` instruction.

Table 4.4: Overview of types of VEX expressions (Section 4.2.4). Some expression types depend on other $\boxed{\textit{VEX expressions}}$ and/or $\langle \textit{parameters} \rangle$.

Textual representation	Value
$\texttt{t}\langle i \rangle$	Read from a temporary with index i.
$\texttt{GET:}\langle \mathit{type} \rangle\texttt{(}\langle j \rangle\texttt{)}$	Read data of specified type from the register with byte offset j in the guest state.
$\texttt{GETI(}\langle \mathit{base} \rangle\texttt{:}\langle n \rangle\texttt{x}\langle \mathit{type} \rangle\texttt{)[}\boxed{ix}\texttt{,}$ $\langle \mathit{bias} \rangle\texttt{]}$	Read from the register at $\mathit{base}+((\mathit{ix}+\mathit{bias}) \bmod n)\cdot s$, where s is the size of *type* (circular indexing).
$\texttt{LDle:}\langle \mathit{type} \rangle\texttt{(}\boxed{\mathit{address}}\texttt{)}$	Read from memory at the given *address*, in little-endian storage order.
$\langle \mathit{op} \rangle\texttt{(}\boxed{q_1}\texttt{, }\boxed{q_2}\texttt{, ...)}$	Operation with one to four arguments, see Tables 4.5 and 4.6 for examples.
$\langle \mathit{literal} \rangle\texttt{:}\langle \mathit{type} \rangle$ or $\langle \mathit{type} \rangle\texttt{\{}\langle \mathit{literal} \rangle\texttt{\}}$	Constant value.
$\texttt{ITE(}\boxed{\mathit{condition}}\texttt{, }\boxed{q_1}\texttt{, }\boxed{q_2}\texttt{)}$	If-then-else construct, selecting either q_1 or q_2 depending on *condition*.
$\langle \mathit{name} \rangle\texttt{\{}\langle \mathit{addr} \rangle\texttt{\}(}\boxed{\mathit{arg1}}\texttt{, }\boxed{\mathit{arg2}}\texttt{,}$ $\texttt{...):}\mathit{type}$	"CCall" to a function *name* in the Valgrind or tool code at *addr*, without any side effects. Arguments are obtained by evaluating *arg1*, *arg2* etc., and the CCall expression evaluates to the return value. Beyond what is shown here, the VEX expression may specify calling convention details and details useful for the Valgrind Memcheck tool.

4.2.4. VEX Expressions

Some of the properties of a statement, such as the index of a temporary of a write, or the pointer and signature of a C function for a dirty statement, are defined with a constant (e. g. of integer or pointer type). In Listing 4.1 and Table 4.2, we have marked such *parameters* in red.

However, most data such as memory addresses, byte offsets of a register, conditions, or inputs to the dirty call, are assigned with a VEX *expression* (`IRExpr`), which are evaluated every time the containing statement executes on the virtual CPU. Table 4.4 gives an overview on the different types of expressions. We have marked expressions with blue frames in Listing 4.1 and Tables 4.2 and 4.4.

Properties of expressions are, again, specified either by parameters or by VEX expressions. The first two of the three temporary-write statements in Listing 4.1 acquire the

Table 4.5: Overview of types of VEX expressions summarized as *operations* in Table 4.4. Continued in Table 4.6.

Textual representation	Value
Scalar floating-point arithmetic	
`AddF64(`rm`,`q_1`,`q_2`)`	Addition of `binary64`s q_1, q_2, using the rounding mode rm.
`MulF32(`rm`,`q_1`,`q_2`)`	Multiplication of `binary32`s q_1, q_2, using the rounding mode rm.
`MAddF64(`rm`,`q_1`,`q_2`,`q_3`)`	Fused multiply-add, computing $q_1 \cdot q_2 + q_3$.
SIMD floating-point arithmetic	
`Mul32Fx8(`rm`,`q_1`,`q_2`)`	Component-wise multiplication of eight `binary32`s in the two `V256` operands, using the rounding mode rm.
Lowest-lane-only SIMD floating-point arithmetic	
`Mul32F0x4(`q_1`,`q_2`)`	Maps the `V128` operands $(q_{10}, q_{11}, q_{12}, q_{13})$ and $(q_{20}, q_{21}, q_{22}, q_{23})$, with four `binary32` components each, to $(q_{10} \cdot q_{20}, q_{11}, q_{12}, q_{13})$.
Floating-point conversions	
`F64toF32(`rm`,`q`)`	Convert `binary64` q to `binary32`, using the rounding mode rm.
`F64toF32(`q`)`	Convert `binary32` q to `binary64`.
Binary reinterpretation	
`ReinterpI64asF64(`q`)`	Change VEX type from `I64` to `F64`, keeping the bitwise data.
`ReinterpF64asI64(`q`)`	Change VEX type from `F64` to `I64`, keeping the bitwise data.
SIMD (un)packing	
`64x4toV256(`q_3`,`q_2`,`q_1`,`q_0`)`	Pack four `I64` arguments into a `V256` SIMD vector (q_0, q_1, q_2, q_3).
`V256to64_2(`q`)`	Extract the `I64` component q_2 from a `V256` SIMD vector (q_0, q_1, q_2, q_2).
`64HIto32(`q`)`	Extract the high `I32` half of an `I64` q.
`32to1(`q`)`	Extract the least-significant bit of the `I32` q.

Table 4.6: Continuation of Table 4.5.

Textual representation	Value
Bitwise logical operations	
`And64(`q_1`,`q_2`)`	Bitwise logical “and” of 64-bit integers q_1, q_2.
Bit-Shifts	
`Shl64(`q`, `n`)`	Bit-shift of an `I64` q by n positions toward the more significant bits, initializing the least significant bit by zero.
`Shr32(`q`, `n`)`	Bit-shift of an `I32` q by n positions toward the less significant bits, initializing the most significant bit by zero.
`Sar32(`q`, `n`)`	Bit-shift of an `I32` q by n positions toward the less significant bits, keeping the most significant bit.
Integer arithmetic	
`Add64(`q_1`,`q_2`)`	Addition of 64-bit integers q_1, q_2.
Comparisons	
`CmpF64(`q_1`,`q_2`)`	The `I32` value is `0x00` if $q_1 > q_2$, `0x01` if $q_1 < q_2$, `0x40` if $q_1 = q_2$, `0x45` if unordered.
`CmpLT64S(`q_1`,`q_2`)`	True if the signed `I64` arguments compare as $q_1 < q_2$, false otherwise.
Miscallaneous	
`SinF64(`rm`,`q`)`	Computes $\sin(q)$ using the rounding mode rm.
`RSqrtEst5GoodF64(`q`)`	“Reciprocal square root estimate, 5 good bits”[168]
`MulD64(`rm`,`q_1`,`q_2`)`	Decimal floating-point multiplication of `D64`s q_1, q_2
`SHA512(`q`,`n`)`	Hash instruction

Listing 4.3: Simple C program to determine whether `binary64` or the 80-bit x87 type is used for `long double` arithmetic.

```c
#include <stdio.h>

int main(){
  long double volatile x = 1.0l;
  x += 0x1.0p-53;
  if( x == 1.0l ){
    printf("x==1, low accuracy\n");
  } else {
    printf("x!=1, high accuracy\n");
  }
}
```

value to be written by an `GET` expression, whose type `I32` and byte offsets `0` and `12` are constant parameters. The right hand side of the third temporary-write statement is a binary operation expression. Its two operands are temporary-read expressions, and the index of the accessed temporary is specified by parameters `3` and `2`, respectively. Finally, the right hand side of the register-write statement is a temporary-read expression as well.

While expressions can be deeply nested, Valgrind only presents flattened superblocks to the instrumentation function (4.2).

4.2.5. Types

Expressions have statically deducible types (`IRType`), such as `I8`, `I16`, ..., `I128` for integers of different sizes; `F32` and `F64` for the IEEE 754 floating-point formats `binary32` and `binary64`; as well as `V128`, `V256` for SIMD data. In statements and expressions accessing memory or the guest state, the number of bytes written or read is determined by the type. Temporaries keep a value of any size, can be assigned once per superblock, and are typed as well. Valgrind performs static type checks on the instrumented VEX code.

There is no general guarantee, however, that the VEX type reflects the “actual” type of data. For instance, when floating-point data is copied around by loading and storing it using an integer type of the same size, as in Listing 3.10, Valgrind would most likely assign an integer type to these data.

4.2.5.a. 80-Bit Type. Note that there is no 80-bit x87 floating-point type in VEX. Valgrind represents the 80-bit registers ST(0), ..., ST(7) by 64-bit sections in the guest state, translates 80-bit floating-point instructions to 64-bit floating-point VEX operations, and emits dirty calls from Table 4.3 to represent the `fldt` and `fstpt` instructions that expose the 80-bit format to memory. Thus, the program in Listing 4.3 prints `x!=0` when run natively, but `x==0` when run under any Valgrind tool.

4.2.5.b. 512-Bit Type. Currently, Valgrind does not support AVX512 instructions with their 512-bit registers, but there are efforts to add such support.[181] As a simple test, we can create a `main` function that calls the function `f` of Listing 3.8, compiling `main` with `-march=skylake` and thus no AVX-512, and linking it with the object file for `f` compiled with AVX-512 support. Natively, the program runs fine on a CPU that supports (the relevant subset of) AVX-512. However, Valgrind 3.18.1, with any tool, prints an error message

```
unhandled instruction bytes: 0x62 0xF1 0xFD 0x48 ↩
    0x10 0x7 0x62 0xF1 0xFD 0x48
```

and raises an illegal instruction signal. Note that the first six bytes encode the AVX-512 `vmovupd` instruction in Listing 3.8.

4.2.6. Example

When Valgrind is invoked with `--trace-flags=11111111` and `--trace-notbelow=0`, it prints a lot of information on each instrumented portion of machine code, including the VEX statements emitted for each machine code instruction, and the entire VEX superblocks before and after instrumentation.

For the assembly language "Hello World" program in Listing 3.14, Valgrind emits one superblock each for either of the two system calls (including the preceding initializations of registers). The lines with white background in Listing 4.4 are the textual representation of the VEX superblock related to the first system call, before instrumentation. We may compare the byte offsets with Table 4.1.

The lines with grey background in Listing 4.4 were added by Valgrind's Memcheck tool (Paragraph 4.1.3.a). For each write to a register at byte offset i in the guest state, Memcheck inserts a zero write of the same size to the byte offset $(i + m_{\text{gs}})$, where $m_{\text{gs}} = 928$ is the total size of the guest state on x86-64 (Table 4.1). Memcheck utilizes these (previously unused) "shadow registers" to keep track of the valid-value bits for the data in the "original registers".

4.3. Advanced Valgrind Features

4.3.1. Monitor Commands

Valgrind's *monitor commands* mechanism enables the user to interact with Valgrind while Valgrind executes the instrumented client program.

4.3.1.a. Mechanism. When Valgrind is started with the command-line argument `--vgdb-error=0`, it activates its built-in "gdbserver" and waits for a connection from a debugger, instead of executing the instrumented client program right away. The user can initiate such a connection e. g. from a GNU Debugger (GDB) session with GDB's `target remote` command. In addition to regular debugger commands like setting breakpoints, stepping, and inspecting memory, the user can then send *monitor commands* to

Listing 4.4: VEX superblock for the `write` system call in Listing 3.14, before instrumentation (lines with white background) and after instrumentation with Memcheck (all lines). Instruction mark statements have been stripped.

```
   PUT(944) = 0x0:I64
   PUT(16) = 0x1:I64
   PUT(1000) = 0x0:I64
   PUT(72) = 0x1:I64
   PUT(992) = 0x0:I64
   PUT(64) = 0x402000:I64
   PUT(960) = 0x0:I64
   PUT(32) = 0xD:I64
   PUT(952) = 0x0:I64
   PUT(24) = 0x40101E:I64
   t1 = CmpNEZ64(0x0:I64)
   DIRTY t1 RdFX-gst(48,8) RdFX-gst(184,8) ::: ↩
       MC_(helperc_value_check8_fail_no_o){0x58010760}()
   PUT(184) = 0x40101E:I64; exit-Sys_syscall
```

Valgrind from the debugger; these commands are passed as a string `req` to functions in the Valgrind core and tool with the signature

```
Bool handle_gdb_monitor_command(ThreadId tid, HChar* req).
```

4.3.1.b. Examples. Valgrind itself defines a few monitor commands, allowing the interactive user to e. g. inspect its internal data structures and status. The Memcheck tool has monitor commands to inspect and modify the addressability and validity bits, and to perform a leak check. As an additional example, the Helgrind tool (Paragraph 4.1.3.c) allows to list all locks and view the access history for any memory address. We will discuss the monitor commands defined by our novel Valgrind tool in Paragraphs 5.4.1.a, 6.5.1.a and 7.3.a.

4.3.1.c. Prerequisites. To allow effective use of the debugger, the client program should contain debug symbols and have been compiled with most optimizations turned off. For the GCC and Clang compilers, this requires the compiler flags `-g` and `-O0` (or `-Og`), respectively.

Debug symbols associate machine code instructions with lines of source code, which is helpful when users want to set a breakpoint at a particular line in order to issue a monitor command when execution reaches this line. Likewise, debug symbols allow to find addresses of variables using their variable names from the source code, and may provide additional information on classes, translation units etc.

Unoptimized builds avoid inlining functions, optimizing out variables, reordering instructions and so on. They are thus more aligned with the mental picture of a human reading the source code.

4.3.2. Client Requests

Valgrind's *client requests* are a "trapdoor mechanism"[168], enabling the instrumented client program to interact with the instrumenting Valgrind core and tool. In contrast to interactively typing monitor commands into a debugger, the user inserts code into the client program that triggers pre-defined actions whenever it executes.

4.3.2.a. Mechanism. In order to specify a client request, the client program has to assemble a data structure, namely, an array of six unsigned 64-bit integers (32 bit on x86). The first entry determines the type of the client request, and the remaining entries transport possible parameters. To perform the request, the client program loads the address of the array into RAX, and then execute a specific sequence of machine code instructions highlighted in Listing 4.5. On a normal x86-64 CPU, this instruction sequence has no effect on memory and registers: First the 64-bit register RDI undergoes a rotation (`rolq`) by $3 + 13 + 61 + 51 = 128$ bits, and then the content of RBX is exchanged (`xchgq`) with itself. When the client is running under Valgrind, however, the pattern is recognized in the instrumentation phase and a special VEX jump statement is emitted instead. In the execution phase, this VEX jump statement leads to a call to client request handlers in the Valgrind core and tool with the signature

```
Bool handle_client_request(ThreadId tid, UWord* arg, UWord* ret)
```

where `arg` is a pointer to the array, and `ret` allows the handler to return information to the client program via the register RDX.

4.3.2.b. Client Request Macros. It is easy to make a client request from an editable C/C++ source, because Valgrind provides header files with preprocessor macros that set up the array describing the client request, and add the specific instruction sequence into the compiled program using the `asm` syntax. Listing 4.5 shows the assembly code produced for the client request macro `VALGRIND_STACK_REGISTER`. Before the marked instruction sequence, the stack is enlarged by 88 bytes (`subq`). This space is used for the array describing the client request, which stores the client request ID `5377` in `16(%rsp)` and the five parameters `200`, `3000`, `0` (default), `0` (default) and `0` (default) in the following addresses. The address of the array is loaded into RAX (with `leaq`). Valgrind calls the client request handler when a normal CPU would execute the marked instruction sequence. Afterwards, the return value of the client request is copied from RDX to the function integer return value register RAX.

4.3.2.c. Examples. Table 4.7 lists examples for client requests provided by the Valgrind core and tools. We will discuss the client requests defined by our novel Valgrind tool in Paragraphs 5.4.1.b, 6.5.1.b and 7.3.b.

4.3.3. Function Wrapping

By Valgrind's *function wrapping* feature, Valgrind tools can have the Valgrind core intercept calls to functions from dynamic libraries, and provide wrappers to which these calls

Listing 4.5: A client request macro in C code, and the assembly code that it expands and compiles to. The special instruction sequence, which indicates a client request to the Valgrind core, has been marked.

```c
#include <valgrind/valgrind.h>

unsigned int f(){
  return VALGRIND_STACK_REGISTER↩
      (200,3000);
}
```

```asm
f:
  endbr64
  subq   $88, %rsp
  movq   %fs:40, %rax
  movq   %rax, 72(%rsp)
  xorl   %eax, %eax
  xorl   %edx, %edx
  movq   $5377, 16(%rsp)
  leaq   16(%rsp), %rax
  movq   $200, 24(%rsp)
  movq   $3000, 32(%rsp)
  movq   $0, 40(%rsp)
  movq   $0, 48(%rsp)
  movq   $0, 56(%rsp)
  rolq $3,  %rdi
  rolq $13, %rdi
  rolq $61, %rdi
  rolq $51, %rdi
  xchgq %rbx,%rbx
  movq   %rdx, 8(%rsp)
  movq   8(%rsp), %rax
  movq   72(%rsp), %rdx
  subq   %fs:40, %rdx
  jne .L5
  addq   $88, %rsp
  ret
.L5:
  call   __stack_chk_fail@PLT
```

Table 4.7: Examples for client request macros defined by the Valgrind core and tools.

Valgrind core:	
`VALGRIND_STACK_REGISTER`	Used as an example here, but flagged as unreliable in the documentation.
`RUNNING_ON_VALGRIND`	Returns `0` if running natively, `1` if running under Valgrind, and larger numbers if running under nested Valgrind layers.
Memcheck:	
`VALGRIND_MAKE_MEM_NOACCESS`	Writes valid-address bits.
`VALGRIND_MAKE_MEM_UNDEFINED`	Writes valid-value bits.
`VALGRIND_MAKE_MEM_DEFINED`	Writes valid-value bits.
`VALGRIND_CHECK_MEM_IS_ADDRESSABLE`	Reads valid-address bits and prints error message if not addressable.
`VALGRIND_CHECK_MEM_IS_DEFINED`	Reads valid-value bits and prints error message if undefined.
Helgrind:	
`ANNOTATE_HAPPENS_BEFORE`, `ANNOTATE_HAPPENS_AFTER`	Can be used to mark thread synchronization constructs that Helgrind does not recognize by itself.

should be rerouted. Listing 4.6 cites an example of such a wrapper from the Valgrind manual[168], which prints the argument and return values of every call to the wrapped function `foo`. The wrapped function has to be specified by its symbol name and, if desired, the "soname" field of the dynamic library that defines it (here, `NONE` specifies no "soname" field). The macro `I_WRAP_SONAME_FNNAME_ZU` encodes these names in the function name of the wrapper. The wrapper must be loaded along with the client program; this can be accomplished without changes to the client executable or libraries, by putting the wrapper into a shared object whose path is in the environment variable `LD_PRELOAD`.

The wrapper function may use standard library functions and client requests, as it is instrumented and executed on the virtual CPU just like the client program.

As the example in Listing 4.6 shows, it is possible to call the original function, obtained via `VALGRIND_GET_ORIG_FN`, from the wrapper function. The macro `CALL_FN_W_WW`, defined in `valgrind.h` and reproduced in Listing B.2, expands to extended `asm` syntax that models a function call according to the appropriate calling conventions: Among other things, it enlarges the stack and moves the two arguments to the registers RDI and RSI, in line with what we saw in Listing 3.1. However, instead of a jump or call instruction, it executes a pattern of four `rolq` and one `xchgq` instruction, similar to the client request pattern in Listing 4.5. This pattern is recognized by the Valgrind core, and translated into a special VEX jump statement which is not subject to redirection by the function wrapping mechanism.

Listing 4.6: Example code for a wrapper of the function `foo`, cited from the Valgrind manual[168].

```
#include <stdio.h>
#include "valgrind.h"
int I_WRAP_SONAME_FNNAME_ZU(NONE,foo)( int x, int y )
{
   int    result;
   OrigFn fn;
   VALGRIND_GET_ORIG_FN(fn);
   printf("foo's wrapper: args %d %d\n", x, y);
   CALL_FN_W_WW(result, fn, x,y);
   printf("foo's wrapper: result %d\n", result);
   return result;
}
```

More details can be found in Appendix B.2. Note that `CALL_FN_W_WW` specifically implements a call to a function with two 64-bit integer arguments and a 64-bit integer return value. Our tool will need to wrap functions with `binary64`, `binary32`, or 80-bit x87 floating-point arguments and return values. As these signatures were not yet supported by the Valgrind framework, we added suitable macro definitions to Derivgrind's version of `valgrind.h`.

4.4. Shadow Memory

Many DBI tools keep track of *shadow data* that stores meta-information about every piece of "original" data handled by the client program. For instance, memory checkers (Paragraph 4.1.3.a) match every byte of data with a valid-value byte (Section 4.2.6). Similarly, taint checkers (Paragraph 4.1.3.b) monitor the boolean information whether data depends on unsafe user input. Our Valgrind tool for algorithmic differentiation will need to store up to two bytes of shadow data for every byte of original data. This section describes how we obtain space for the shadow data.

4.4.a. Shadow Registers. In the VEX superblocks created from the client program's machine code and presented to a Valgrind tool for instrumentation, registers of the real CPU map to well-defined byte offsets in the guest state. On x86-64, part of the mapping is shown in Table 4.1, and in total $m_{\text{gs}} = 928$ bytes of the guest state are used (in Valgrind 3.18.1, which does not support AVX-512 registers so far). However, the guest state of Valgrind's virtual CPU is actually three times as large. Thus, Valgrind tools may use registers at byte offsets $j + m_{\text{gs}}$ and $j + 2m_{\text{gs}}$ to store up to two bytes of shadow data per byte of original data at byte offset j. In Section 4.2.6, we saw that Memcheck indeed uses one layer of shadow registers by shifting byte offsets by m_{gs}.

4.4.b. Shadow Temporaries. Each VEX superblock also has an exclusive upper bound m_{tmp} for the indices of temporaries. The number of available temporaries is comparatively large, so typically the temporaries $i + m_{\text{tmp}}$ and $i + 2m_{\text{tmp}}$ can be used to store shadow data for the data in temporary i. We observed two kinds of error messages in rare cases where this led to problems:

- "VEX temporary storage exhausted. Increase `N_{TEMPORARY,PERMANENT}_BYTES` and recompile": This problem disappeared after increasing the two constants in the source code of Valgrind.

- "Assertion '`instrs_in->arr_used <= 15000`' failed": When this problem shows up (as e. g. in Paragraph 8.2.e), it can be fixed by supplying an additional argument

  ```
  --vex-guest-max-insns=10
  ```

  (or an even smaller number) to the `valgrind` call (4.1).

4.4.c. Shadow Memory. For memory, the analogous "index shifting" approach would mean that shadow data is stored at memory locations with fixed offsets from the memory address of the original data. Note that glibc's `malloc`, and thus most Linux userspace processes, mainly use the `mmap` system call to allocate memory. `mmap` requests the kernel to map sections of the virtual address space of the process to physical memory (Section 3.2.6). While processes can suggest a start address of the new mapping in the virtual address space, the kernel is free to assign other areas of the virtual address space. Thus, DBI tools cannot reliably allocate shadow memory at a particular virtual address, but this would be necessary for a simple address-shifting approach.

A much more robust approach is to resort to *shadow memory tools*[45,138] in order to keep track of shadow data for the memory used by the client program. A shadow memory tool acts like an associative array that maps memory addresses to the corresponding shadow data.

4.4.d. Implementation of Shadow Memory. A popular way of implementing shadow memory[138] uses a data structure similar to multilevel page tables for virtual memory (Section 3.2.6) and related to prefix trees (tries). This is illustrated in Figure 4.2. When virtual addresses have a length of $k = k_1 + \cdots + k_n$ bits (here: $n = 3$, $k_1 = 3$, $k_2 = 2$, $k_3 = 2$) and the shadow data for one address has a size of N bytes, the shadow data for a given virtual address (here: `0x1010010`) is looked up as follows.

- A single *primary map* (indicated in red) stores 2^{k_1} (here: $2^3 = 8$) pointers (indicated by empty and filled circles). The highest-significant k_1 bits of the virtual address (here: `101`) are used as an index (here: `5`) to retrieve a pointer to ...

- ... a *secondary map* (indicated in blue), which stores 2^{k_2} (here: $2^2 = 4$) pointers (indicated by empty and filled circles). The next-most-significant k_2 bits of the virtual address (here: `00`) are used as an index (here: `0`) to retrieve a pointer to ...

- ... (more maps might follow, but there are none in this example) ...

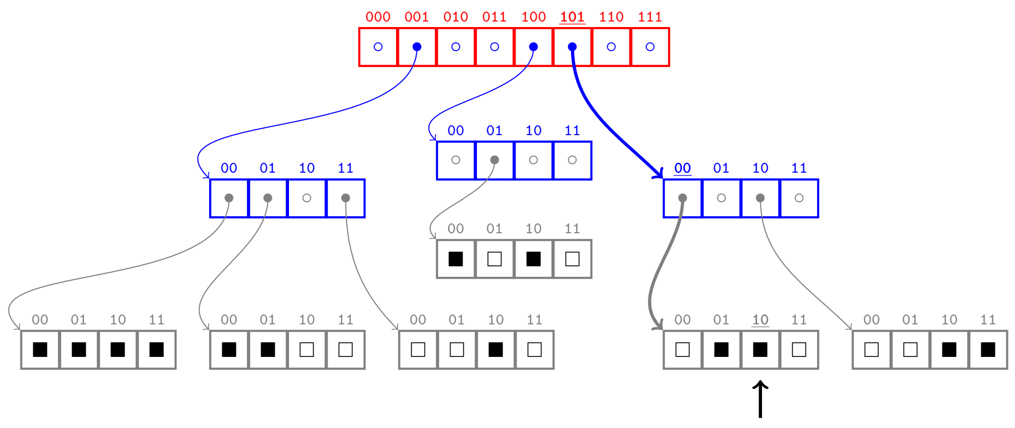


Figure 4.2: Shadow memory implementation using a two-level page table, with a primary map resolving the most-significant three bits of an address, and secondary maps resolving two more bits. Leaves store shadow data, indicated by (empty or filled) black squares, for four consecutive addresses each. The marked black square is the shadow data for the address `0x1010010`.

- . . . a *leaf* (or *page*; indicated in gray), which stores 2^{k_n} (here: $2^2 = 4$) shadow objects (indicated by empty and filled black squares), occupying N bytes each. The least-significant k_n bits of the virtual address (here: `10`) are used as an index (here: 2) to retrieve the shadow data (indicated by the black vertical arrow) for the given virtual address.

The internal data structures of the shadow memory tool, like the above maps and leaves, are stored in virtual memory, possibly next to the data for which they provide shadow storage. In the end of each lookup, the shadow memory tools might give access to the shadow data either via getter and setter functions, or by returning the memory address. Note that while technically possible, there is normally no reason for the user of a shadow memory tool (like a Valgrind tool) to have it shadow its own internal datastructures.

All maps except for the primary map, and all leaves, are lazily allocated when there is a write operation in their respective address range of the shadow memory. Pointers in maps are initialized with null pointers when the respective child map or leaf has not yet been allocated (indicated by empty circles in Figure 4.2). Entries of a leaf are initialized with a default shadow data value specified by the user of the tool (indicated by empty squares). When a null pointer is encountered while trying to read from the shadow memory (as it would happen for the address `0x0011010` in Figure 4.2), the default shadow value (or a constant reference to a static default leaf) is returned. If the original data is not too fragmented, this ensures that the total memory consumption of the shadow memory tool is about proportional to the amount of original data, with a factor not much worse than N.

4.4.e. Choice and Design of a Shadow Memory Tool. In the beginning, our Valgrind tool relied on an existing shadow memory tool by Cronburg[45], which was easy to integrate and could be configured to use the custom memory allocation functions of Valgrind – this is important because Valgrind tool code must not depend on the C standard library. Later on, we developed and integrated a novel shadow memory tool that has been taylored for our application in the following ways.

- The tool by Cronburg[45] provided access to shadow data only via getter and setter functions. This is an inefficient interface when the shadow data at multiple consecutive addresses is frequently accessed together, because the page tables must be traversed repeatedly for every single address. Our shadow memory implementation returns pointers to the shadow data in the leaf, so unless the access crosses a leaf boundary, only a single look-up is performed.

- We required a more flexible choice of the number N of shadow bytes per original byte than what the tool by Cronburg[45] offered.

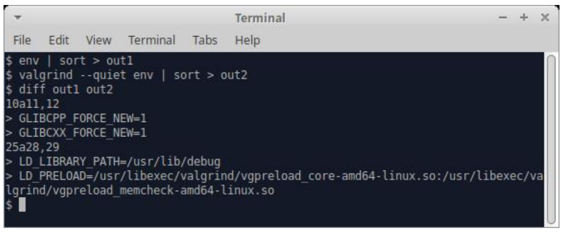
- For 64-bit addresses, the primary map of the tool by Cronburg[45] occupied 4 GB. Initializing it usually takes more than a second; also, 4 GB was already above the tight memory constraints of our continuous integration (CI) environment. The initial memory consumption can be massively reduced by moderately increasing the number of shadow map levels and reducing the numbers of address bits resolved by each of these levels. Our shadow memory implementation makes such a choice very easy at the compile time of the Valgrind tool using it.

4.5. Limitations

Valgrind is a robust and versatile framework for dynamic binary instrumentation and we use it to insert automatic differentiation logic into compiled programs. Besides the limitations of machine-code-based AD in general, as outlined in Section 3.3, users have to keep DBI- and Valgrind-specific limitations in mind. We discuss the technical aspects in Section 4.5.1 and sketch the legal aspects in Section 4.5.2.

4.5.1. Technical Limitations

4.5.1.a. Performance Degradation. “Heavyweight” Valgrind tools, which add instrumentation of a complexity comparable to the original code and involve shadow memory, are known to incur a significant run-time slow-down of the client program.[139,168] The run-time typically scales by factors ranging from 10 to 100, depending on the tool and the client program; for the Memcheck tool with an optimized shadow memory handling, a mean slow-down factor of 22 has been reported.[139] Performance is only one out of multiple design goals of AD tools (Paragraph 1.1.c), and low performance is usually acceptable for exploratory studies, or even in production if there is no alternative.



```
$ env | sort > out1
$ valgrind --quiet env | sort > out2
$ diff out1 out2
10a11,12
> GLIBCPP_FORCE_NEW=1
> GLIBCXX_FORCE_NEW=1
25a28,29
> LD_LIBRARY_PATH=/usr/lib/debug
> LD_PRELOAD=/usr/libexec/valgrind/vgpreload_core-amd64-linux.so:/usr/libexec/va
lgrind/vgpreload_memcheck-amd64-linux.so
$ 
```

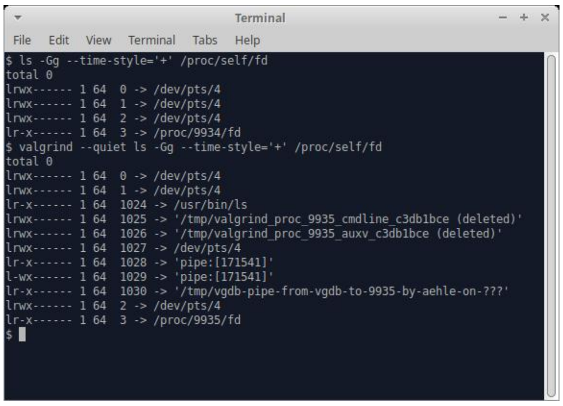
Figure 4.3: The program `env` prints all environment variables to standard output. As we can see, Valgrind (system installation of version 3.18.1 on Ubuntu 22.04.3) modifies the environment of the client program; e. g., two shared libraries are added to `LD_PRELOAD`.

4.5.1.b. Incomplete Transparency to the Client Program. Apart from running slower, client programs should execute in the functionally same way under DBI tools as they do on a CPU; except, of course, for places where the user intervened explicitly, e. g. by inserting a client request. However, even disabling client requests, function wrapping etc. will not make the actions of the DBI tool fully transparent to client programs if they pay close attention. The cybersecurity community has identified a wide variety of *evasion techniques* by which malware can discover that it runs under a DBI framework, e. g. with the purpose of obstructing analyses by changing its behavior.[59] For example, the client program can scan the entire virtual address space for artifacts of a DBI framework, search for clues in the environment variables (Figure 4.3), look for additional file descriptors (Figure 4.4), measure whether its run-time or memory performance has degraded, or employ one of the technical limitations listed further down. See Bruening et al.[39] for a variety of other transparency issues and possible solutions in the context of DynamoRIO.

We believe that "adversarial" client programs are a specialty of cybersecurity research, and that most software from other scientific or industrial domains is not specifically designed to cause trouble under DBI frameworks.

4.5.1.c. Floating-Point Accuracy. As outlined in Paragraph 4.2.5.a, Valgrind replaces 80-bit by 64-bit floating-point arithmetic. In most cases, the reduced floating-point accuracy will have a minor effect on the output values computed by the program, if at all. However, Listing 4.3 demonstrates that in principle, a small change in an intermediate result can flip the outcome of a floating-point comparison, which might subsequently alter the entire control flow of the program, leading to a completely different output. We believe that this scenario is rather hypothetical for the class of client programs that AD is typically applied to, because if the value and/or derivative of a function is very sensitive with respect to the input, its derivatives likely have little value for applications anyway.

4.5.1.d. Unrecognized Instructions. As furthermore outlined in Paragraph 4.2.5.b, Valgrind cannot translate AVX-512 instructions into VEX IR at the moment. Until such support is added, the Valgrind framework cannot instrument client programs involving



```
$ ls -Gg --time-style='+' /proc/self/fd
total 0
lrwx------ 1 64  0 -> /dev/pts/4
lrwx------ 1 64  1 -> /dev/pts/4
lrwx------ 1 64  2 -> /dev/pts/4
lr-x------ 1 64  3 -> /proc/9934/fd
$ valgrind --quiet ls -Gg --time-style='+' /proc/self/fd
total 0
lrwx------ 1 64  0 -> /dev/pts/4
lrwx------ 1 64  1 -> /dev/pts/4
lr-x------ 1 64  1024 -> /usr/bin/ls
lrwx------ 1 64  1025 -> '/tmp/valgrind_proc_9935_cmdline_c3db1bce (deleted)'
lrwx------ 1 64  1026 -> '/tmp/valgrind_proc_9935_auxv_c3db1bce (deleted)'
lrwx------ 1 64  1027 -> /dev/pts/4
lr-x------ 1 64  1028 -> 'pipe:[171541]'
l-wx------ 1 64  1029 -> 'pipe:[171541]'
lr-x------ 1 64  1030 -> '/tmp/vgdb-pipe-from-vgdb-to-9935-by-aehle-on-???'
lrwx------ 1 64  2 -> /dev/pts/4
lr-x------ 1 64  3 -> /proc/9935/fd
$
```

Figure 4.4: The program `ls` is used to show the contents of the directory `/proc/self/fd` on the pseudo-filesystem `proc`, which contains the open file descriptors of the process that accesses it. Normally, these are the standard input, output and error streams (here, connected to the pseudo-terminal `/dev/pts/4`) and any other file descriptors opened (here, the `fd` directory whose contents `ls` shows). When running under Valgrind (system installation of version 3.18.1 on Ubuntu 22.04.3), the client program sees additional file descriptors.

these instructions. AVX instructions are supported on x86-64 but not on x86. Also, 3DNow! instructions are not supported by Valgrind according to the manual,[168] but they are deprecated and very rarely used anyhow.

4.5.1.e. Only userspace code is instrumented. Valgrind instruments the entire userspace code including system libraries. Valgrind tools can therefore observe and manipulate the entire interaction between the userspace code and the operating system, e. g. via system calls (Section 3.2.4). However, this wrapping must be implemented as part of the Valgrind tool, as Valgrind cannot instrument the operating system kernel. This is why, e. g., special care is needed to use Valgrind-Memcheck for MPI-parallel programs. We cannot expect that the entire system call API is perfectly wrapped by the Valgrind core and/or tool.

For example, when we replace the constant `42` in Listing 3.15 by some undefined variable and run the code under Valgrind-Memcheck, no warning message is produced, indicating that our installation of Valgrind-Memcheck (version 3.18.1) is not aware of the dependency between `*x` and `*y` created using the `mmap` system call.

4.5.2. Legal Limitations

Most jurisdictions around the world have developed a concept of *intellectual property* (IP): Rights related to “intellectual goods” such as inventions, artwork and trademarks can be owned, and within certain extents, the owner decides how these goods may be used.[194] Software, both in the form of source code and machine code, is mainly protected by *copyright law* in a very similar fashion as literary and artistic creations;[25,191] sometimes, *patent law*, which protects technical inventions, can also be relevant.[54,170] Users of software must respect the rights of the owners of the software, who may prohibit certain ways of dealing with their software that would be possible from the technical side. This section summarize a layman's perspective on such legal limitations; we are not lawyers and this is no legal advice.

4.5.2.a. Licenses. Generally, it is not allowed (or at least legally risky) to run, modify, or distribute software without the permission of the copyright owner. Typically, users obtain a *license* granting them a subset of these rights under certain conditions. For instance, many licenses disclaim any warranties and liabilities of the granting party. *Proprietary licenses* formulate a rather narrow set of rights. In contrast, licenses in the open-source world base the receiving party's rights on one of the following ideas:

- *Permissive licenses*, like the MIT or BSD-style licenses, basically allow any kind of use, modification and distribution of the software, with minor provisions concerning proper attribution and conservation of copyright notices.
- *Copyleft licenses*, like the GNU General Public License (GPL)[62,63] in version 2 or 3, permit use and modification. Furthermore, for software received in compiled form, it grants the right to obtain the source code. However, distribution is only allowed under similar license terms, and this *copyleft clause* also "infects" derived and combined works. The main goal behind this license construct is to ban the use of GPL-licensed software in software distributed under proprietary licenses, in order to give a competitive advantage to open-source software projects and to keep them from becoming proprietary in the future.
- *Weak copyleft licenses*, like the GNU Lesser General Public License (LGPL)[64,65], work similarly to copyleft licenses, but do not "infect" other parts in combined works; this allows their use in proprietary projects, and still helps to keep them from becoming proprietary.

When DBI tools are applied to a client program, the user must simultaneously comply with the license terms of the DBI framework, the DBI tool, and the client program.

4.5.2.b. Restrictions by the DBI Framework/Tool. Valgrind is available under the GPLv2 license. Valgrind tools are closely linked with the Valgrind framework, meaning that they must be distributed under the terms of the GPLv2 as well. A lot of code in the Valgrind distribution, and our changes and additions to implement the Derivgrind AD tool, are actually licensed under the GPLv2 "or any later version".

Valgrind's GPL license does not prohibit the application of Valgrind to proprietarily licensed client programs. Even if the machine code produced by Valgrind were considered a derived work – which it is not in the somewhat similar case of software built by GCC –, the instrumented machine code only exists temporarily in the main memory of the user's computer and is not distributed. Thus, the restrictions on distribution imposed by the GPL after any potential copyleft infection would be anyhow irrelevant.

There is one exception to the previous statements: Valgrind's client request macros are statically compiled into the client program. For this reason, Valgrind's client request headers have been separately licensed under a permissive BSD-style license, so generally, they can be freely included without major licensing implications.[168] Likewise, Derivgrind's client request headers come under a permissive MIT license.

DynamoRIO is available under a BSD license. PIN is available under proprietary licenses; commercial users and tool developers must thus pay special attention to the applicable terms and conditions.

4.5.2.c. Restrictions by the Client Program. DBI tools disassemble and modify portions of the client program. This should be fine if the client program was obtained under an open-source license. In the case of proprietary client programs, we recommend to study the license terms before running them under DBI tools, to see whether these actions are covered, and to consult a lawyer in case of doubt.

5. Forward-Mode Automatic Differentiation of Compiled Programs

In the first part of this dissertation, we have given an overview on algorithmic differentiation (Chapter 2) and dynamic instrumentation (Chapter 4) of machine code (Chapter 3). We will now assemble these “building blocks” into an AD tool applicable to compiled computer programs.

This chapter is about the key ideas and aspects to consider regarding the forward mode of AD. As outlined in Section 2.3, forward-mode AD logic stores dot values for all real numbers handled during the execution of the program, and propagates them alongside the real-arithmetic operations performed by the primal program. We therefore discuss the storage format (Section 5.1), processing (Sections 5.2 and 5.3) and user access mechanisms (Section 5.4) of dot values. The math function wrappers introduced in Section 5.5 compensate for bit-tricks (Section 3.3) in the C math library.

Derivgrind’s recording capabilities follow similar ideas, but differ in many details, which we elaborate on in the next Chapter 6. A third kind of instrumentation, a heuristic to detect bit-tricks, will be presented in Chapter 7.

5.1. Shadow Data: Dot Values

As outlined in Section 4.4, the Valgrind framework and our shadow memory tool provide a Valgrind tool with access to two shadow storage locations for each VEX temporary, register and memory address, of the same size and type. For its forward-mode capabilities, Derivgrind uses only one layer of shadow memory, in the following way.

5.1.a. Shadowing Policy. Whenever a set of bytes in a storage location stores parts of a binary representation of a floating-point value a in any floating-point format, Derivgrind’s instrumentation should make sure that the same set of bytes in the shadow location stores the same parts of the representation of the dot value $\dot{a}$, in the same format. An example of this is shown in Figure 5.1. When bytes in a memory location do not originate from, and are not further used as floating-point data, the corresponding bytes in the shadow location are unspecified.

Suppose, for instance, that a 64-bit temporary $\texttt{t}\langle i \rangle$ is initialized by joining together two floating-point representations as displayed in Figure 5.2: In the lower half, the lower four bytes of the `binary64` in Figure 5.1, and in the upper half, a `binary32` constant 1.0. Then, the lower four bytes of the shadow temporary $\texttt{t}\langle i + m_{\text{tmp}} \rangle$ should be initialized with the lower four bytes of the `binary64` representation of $\dot{a}$, and the upper four bytes should be the dot value of a constant, $0.0 =$ `0x00000000`.

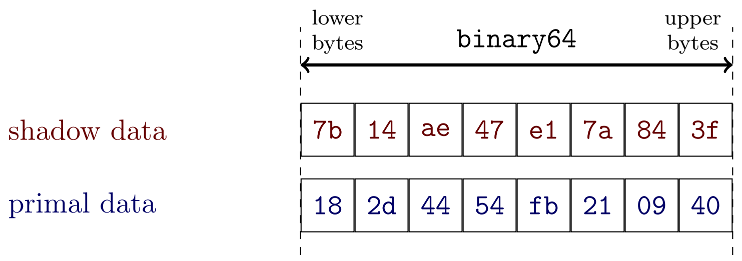


Figure 5.1: The dot value $\frac{1}{100}$ = `0x3f847ae147ae147b` is stored in the shadow memory (red) over the `binary64` number `0x400921fb54442d18` (blue, representing the value $\pi \approx 3.14$).

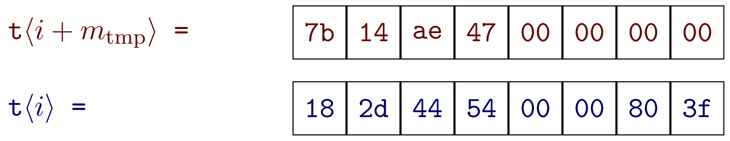


Figure 5.2: Suppose that the four-byte temporary $\texttt{t}\langle i\rangle$ is assigned with the lower four bytes of the `binary64` data in Figure 5.1 followed by a constant `binary32` 1.0 (`0x3f800000`). According to Paragraph 5.1.a, the shadow temporary $\texttt{t}\langle i + m_{\text{tmp}}\rangle$ (red) stores the respective parts of the floating-point representations of the dot values.

5.1.b. Rationale. We arrived at this exact form of the policy from the following considerations:

- A weaker form of this policy could demand that the shadow storage location stores the dot value only for expressions of floating-point VEX types (such as `F32` and `F64`). However, this would lead to incorrect results when floating-point data is transferred between memory locations in a type-agnostic fashion using integer types, as in Listing 3.10. Moreover, for the SIMD types `V128` and `V256`, there is no distinction at all between integer and floating-point types.

- With SIMD registers sometimes being used to store a single floating-point value (usually in their lowest 32-bit or 64-bit lane, as in Listing 3.7), it is obvious that the policy should refer to *sets of bytes in storage locations* rather than entire storage locations.

- Floating-point representations can be split and copied in multiple portions (e. g. first the lower 32-bit half of a `binary64`, then the upper 32-bit half). This is why the policy refers to *parts of a binary representation* rather than entire binary representations.

- Using the *same floating-point format* for the value and the dot value is a practical choice. It would be possible to store `binary64` dot values for `binary32` values using two shadow memory layers; actually, our recording-mode shadow data policy, described in Paragraph 6.1.b, succeeds in keeping track of 64-bit indices for 32-bit floating-point numbers this way. Here, we prefer a simple solution and store `binary32` dot values for `binary32` values.

5.1.c. Default Dot Values. Note that the type of global variables and constants in the data and bss sections of a program is not known a priori. Our policy mandates that for variables and constants of floating-point type, their shadow is a floating-point zero of the same type. This can be easily achieved by setting the default value of uninitialized shadow memory to a bitwise zero, which represents +0.0 in the `binary32` and `binary64` formats as well as in the x87 extended precision format.

5.1.d. Summary of the Instrumentation. In the following Sections 5.2 and 5.3, we describe the forward-mode, i. e., *dot-value-propagating* AD logic that Derivgrind inserts into the VEX code of the client program. It updates the shadow temporaries, registers and memory in such a way that their content always complies with the shadowing policy (Paragraph 5.1.a). The instrumentation is based on the following two main facts:

- The instrumentation of data transfers between two storage locations can simply transfer the corresponding shadow data between the two respective shadow storage locations.
- The instrumentation of real-arithmetic operations computes the dot value of the result from the values and dot values of the operands, according to the appropriate differentiation rules.

5.2. Instrumentation of Statements

5.2.1. General Case

In front of all types of VEX statements with relevance to AD, except for CAS statements, the Derivgrind tool inserts *dot-value-propagating statements.* Examples of dot-value-propagating statements are shown in Table 5.1, partially repeating information from Table 4.2 for the convenience of the reader. We further discuss two special cases in Sections 5.2.2 and 5.2.3 below. When a statement involves an expression p that can evaluate to a representation of a real number, the dot-value-propagating statement may involve the *dot-value-propagating expression* $\dot{p}$ that evaluates to the representation of the dot value, and is constructed as described in the next Section 5.3. Sometimes, forming the dot-value-propagating expressions requires to insert additional VEX statements in front.

5.2.2. Compare-and-Swap Statements and Multi-Threading

AD of multi-threaded code, mostly via OpenMP, has been successfully implemented in multiple source-code-based AD tools.[30,32,40,84,130] However, Derivgrind does not fully

Table 5.1: Augmentation of VEX statements with forward-mode AD logic.

VEX statement	Additional VEX statements for forward-mode AD
Store x in a temporary with index i. `t`$\langle i\rangle$ `=` x	Store $\dot{x}$ in the shadow temporary. `t`$\langle i+m_{\mathrm{tmp}}\rangle$ `=` $\dot{x}$
Store x in the register with byte offset j in the guest state. `PUT(`$\langle j\rangle$`)` `=` x	Store $\dot{x}$ in the shadow register. `PUT(`$\langle j+m_{\mathrm{gs}}\rangle$`)` `=` $\dot{x}$
Store x in memory. `STle(`*address*`)` `=` x	Store $\dot{x}$ in shadow memory. Implemented as a dirty call to access the shadow memory from VEX.
Store x in memory, if a condition is satisfied. `if (`*guard*`) STle(`*address*`)` `=` x	Store $\dot{x}$ in shadow memory, subject to the same condition. Implemented as a dirty call.
Compare-and-swap, loading *addr* into temporary `t`$\langle$*old*$\rangle$, and replacing data at *addr* by *new* if it matches *expd*. `t`$\langle$*old*$\rangle$ `= CASle(`*addr* `::` *expd* `->` *new*`)`	CAS statements are replaced as discussed in Section 5.2.2.
Dirty call, invoking a Valgrind function with side effects.	The augmentation depends on the dirty call, see details in Section 5.2.3.
Meta information. `------ IMark(...) ------`	Not relevant for AD.
Conditional jump. `if (`*guard*`) goto {`$\langle$*jump kind*$\rangle$`}` $\langle$*target*$\rangle$	Not relevant for AD.

support multi-threading, for the reasons described in the following.

5.2.2.a. Valgrind and Multi-Threading. The Valgrind framework supports multi-threaded client programs, but has an internal lock that only allows one thread to run at a time. Thus, a multi-threaded variant of a client program will generally not finish faster than a serial variant when run under Valgrind. Nevertheless, support for multi-threaded client programs can be considered an asset, because it allows users to apply Valgrind tools without worrying about e. g. OpenMP constructs in the code.

However, we should note that high-level multi-threading constructs are generally much easier to spot on the source code level than in VEX. For example, the OpenMP `#pragma omp atomic` construct compiles to a sequence of instructions as shown in Listing 3.13, which themselves may translate to multiple VEX statements as shown in Listing 4.2. One of these VEX statements is a CAS statement, but other statements, such as a comparison to check if the CAS was successful, are equally important.

5.2.2.b. Derivgrind instruments the CAS itself. For the correct forward-mode AD handling of CAS statements, it is important to realize that the shared state of the forward-AD-augmented program consists of both the value of the shared state in the original program (i. e. `*addr` in Listing 3.13), and its dot value in the corresponding shadow location. Due to their AD augmentation, threads will always update both, but it can happen that an update leaves either the value or the dot value unchanged. To determine whether the shared state has changed, it is therefore mandatory to compare both the present value at `addr` with the expected value, and the corresponding dot values.

For this reason, we replace a CAS instruction by a construct that first performs the two comparisons, and only if they both succeed, writes to memory and to shadow memory. It is not a problem for Derivgrind that this construct is non-atomic, because Valgrind runs only one thread at a time and prevents context switches in the middle of an instrumented instruction.

5.2.2.c. Derivgrind cannot robustly instrument the check whether a CAS was successful. Aside from the CAS statement, Listing 4.2 contains another comparison by means of the `amd64g_calculate_condition` CCall. Without further care, this function would determine equality only based on values, not taking dot values into account. We discuss wrapping of VEX expressions in the next Section 5.3, but can already note here that it is not a good idea to modify generic checks for floating-point equality. Doing so would, for example, also affect statements like `if(x==0.0)`, which should be `true` if the value of `x` is zero even if the dot value is not.

And even if one managed to modify only those `amd64g_calculate_condition` CCalls which originate from translating `lock cmpxchgq` and related instructions to VEX, it would not be a full solution. Other compiler versions or flags may insert a separate comparison to decide whether the CAS was successful. We have seen one example for this in Listing 3.22 (lines highlighted in yellow), using bitwise logical operations for the comparison (note that they also affect the zero flag controlling the conditional jump instruction `jne`). It is hard to recognize, in an automatic fashion, that a comparison is performed there. Additionally, it would be hard to find out that, semantically, this comparison checks if a shared state has been modified and must thus also take dot values into account.

As a consequence, Derivgrind's AD logic may introduce data races into multi-threaded client programs.

5.2.3. Dirty Call Statements

For the dirty calls converting between `binary64` and the 80-bit x87 floating-point type (see Table 4.3), Derivgrind inserts dirty calls that perform the analogous operation on the shadow data, to comply with our policy to use the same floating-point format for values and dot values (Section 5.1).

As far as we observed it, all the other dirty calls in Table 4.3 can hardly be part of a floating-point calculation and therefore do not require any specific AD logic. For dirty calls that write to an output temporary, the respective shadow temporary is assigned with a value of zero, so it is defined in case it is later read from.

5.3. Instrumentation of Expressions

Many of the dot-value-propagating statements in Table 5.1 and Sections 5.2.2 and 5.2.3 involve a dot-value-propagating expression $\dot{p}$ computing the dot value of the expression p. Derivgrind forms dot-value-propagating expressions according to Table 5.2. Table 5.3 gives more details for the important subclass of expressions that perform an operation. Information from Tables 4.4 to 4.6 is partially repeated here for the convenience of the reader.

5.3.1. Bitwise Logical Operations

Among the variety of bit-tricks listed in Section 3.3, Derivgrind supports manipulations of the sign bit (Section 3.3.2) and masking of complete floating-point representations (Section 3.3.3) for aligned floating-point representations. To this end, Derivgrind instruments VEX expressions for the bitwise logical “and”, “or” and “exclusive-or” operations. For example, the dot-value-propagating expression for a bitwise “or” works in the following way:

(a) First, operands of type `V128` and `V256` are split into `I64` blocks. For simplicity, `I32` operands are zero-padded into `I64` blocks as well. In the following, we refer to the `I64` operands as p and q, and to their dot values as $\dot{p}$ and $\dot{q}$.

(b) If p is equal to the 64-bit constant `0b10...0` and $\dot{p}$ is equal to `0b0...0`, Derivgrind assumes that this expression computes the negative absolute value of q. Thus, the dot value of the expression is $-\dot{q}$ if $q > 0$, and $\dot{q}$ otherwise.

Table 5.2: VEX expressions, and construction of the corresponding dot-value-propagating expressions.

Expression p	Dot-value-propagating expression $\dot{p}$
Read from a temporary with index i. `t`$\langle i \rangle$	Read from the shadow temporary. `t`$\langle i + m_{\text{tmp}} \rangle$
Read data of specified type from the register with byte offset j in the guest state. `GET:`$\langle type \rangle$`(`$\langle j \rangle$`)`	Read data of the same type from the shadow register. `GET:`$\langle type \rangle$`(`$\langle j + m_{\text{gs}} \rangle$`)`
Read from memory. `LDle:`$\langle type \rangle$`(`*address*`)`	Read from shadow memory, expecting data of the same type. Implemented as a dirty call.
Operation with one to four arguments, see Table 5.3 for examples. $\langle op \rangle$`(`q_1`, `q_2`, ... )`	See Table 5.3.
Constant value. $\langle literal \rangle$`:`$\langle type \rangle$ or $\langle type \rangle$`{`$\langle literal \rangle$`}`	Constant value zero of the same type. `0x0:`$\langle type \rangle$ or $\langle type \rangle$`{0x0}`
If-then-else construct, selecting either a or b depending on *condition*. `ITE(`*condition*`, `q_1`, `q_2`)`	If-then-else construct with same condition on dot-value-propagating operands. `ITE(`*condition*`, `$\dot{q}_1$`, `$\dot{q}_2$`)`
CCall to Valgrind function without side effects.	Until now, we only encountered cases without relevance to AD.

(c) Otherwise, if p and $\dot{p}$ are both equal to the 64-bit constant `0b0...0`, Derivgrind assumes that this expression belongs to a masking operation like (3.3) that selects q. Thus, the dot value of the expression is $\dot{q}$.

(d) Otherwise, the checks (b), (c) are performed with swapped roles of p and q.

(e) If none of the previous cases applied, analogous checks to (b), (c) and (d) are performed separately on both 32-bit halves of p and q. In either half, if no supported bit-trick is recognized, the dot value of the expression is a 32-bit `0b0...0`.

In steps (b) and (c), the dot value $\dot{p}$ is compared to zero because apart from appearing as non-floating-point operands to compute the negative absolute value or perform a masking patterns, `0b10...0` and `0b0...0` can also represent valid real numbers -0.0 and $+0.0$, respectively. If p represents a real number (thus likely with $\dot{p} \neq 0.0$) and q is the constant `0b10...0` or `0b00...0` used to perform a bit-trick on p (thus likely $\dot{q} =$ `0b0...0`), we must not confuse the roles of p and q, and vice versa.

Dot-value-propagating expressions for bitwise “and” and “exclusive-or” operations are formed in a very similar fashion. For the bitwise “and”, confusing the non-floating-point operand with the floating-point operand in (b) and (c) is hardly possible because p is compared with `0b01...1` and `0b1...1`, respectively, which both represent NaN. The bitwise “exclusive-or” does not appear in the masking pattern. It can be used to create a zero value (by applying the “exclusive-or” of a value with itself), but in this case, the default dot value of `0b0...0` is correct.

Sign bit manipulations and masking patterns outside of aligned 32-bit and 64-bit blocks, and all the other bit-tricks presented in Section 3.3, are not supported by this approach of wrapping bitwise logical operations.

5.3.2. Unhandled Expressions

From the large number of operations available in VEX (from which only a selection is shown in Tables 4.5 and 4.6), we only handle those that we consider necessary and that we suspect to be covered by our unit tests. For instance, the VEX operation `SinF64` corresponding to the x87 instruction `fsin` is not handled at the time of writing this sentence because apparently, modern compilers and libraries do not use this instruction to compute the sine function. For unhandled operations, Derivgrind assumes a zero derivative, and optionally outputs warning messages according to the optional Derivgrind argument `--warn-unwrapped=`⟨*warnlevel*⟩:

- With `yes` or `all`, Derivgrind prints information about all statements containing unhandled expressions.
- With `default` or if the argument is not set, Derivgrind warns about statements that contain unhandled expressions that have a floating-point return type and at least one operand of floating-point type.
- With `no` or `none`, Derivgrind does not print warning messages about unhandled expressions.

Table 5.3: VEX IR expressions performing an operation, and the corresponding dot-value-propagating expressions. The placeholder rm represents an expression for the rounding mode, for which a dot-value-propagating expression is never required.

Expression p	Dot-value-propagating expression $\dot{p}$

Scalar floating-point arithmetic, e. g. addition of `binary64`s,
`AddF64(`rm`,`q_1`,`q_2`)`
or multiplication of `binary32`s,
`MulF32(`rm`,`q_1`,`q_2`)`
	Application of the differentiation rule.
	
	`AddF64(`rm`,`$\dot{q}_1$`,`$\dot{q}_2$`)`
	
	`AddF32(`rm`,MulF32(`rm`,`$\dot{q}_1$`,`q_2`),`
	`MulF32(`rm`,`q_1`,`$\dot{q}_2$`))`

SIMD floating-point arithmetic, e. g. multiplication of eight `binary32`s.
`Mul32Fx8(`rm`,`q_1`,`q_2`)`
	Component-wise application of the differentiation rule.
	`Add32Fx8(`rm`,Mul32Fx8(`rm`,`$\dot{q}_1$`,`q_2`),`
	`Mul32Fx8(`rm`,`q_1`,`$\dot{q}_2$`))`

Lowest-lane-only SIMD floating-point arithmetic, e. g. mapping the operands $(q_{10}, q_{11}, q_{12}, q_{13})$ and $(q_{20}, q_{21}, q_{22}, q_{23})$ to $(q_{10} \cdot q_{20}, q_{11}, q_{12}, q_{13})$.
 	
`Mul32F0x4(`q_1`,`q_2`)`
	Formal application of the differentiation rule with lowest-lane-only operations, taking care to be correct outside the lowest lane also. E. g. for $(\dot{q}_{10}q_{20} + q_{10}\dot{q}_{20}, \dot{q}_{11}, \dot{q}_{12}, \dot{q}_{13})$,
	`Add32F0x4(Mul32F0x4(`$\dot{q}_1$`,`q_2`),`
	`Mul32F0x4(`q_1`,`$\dot{q}_2$`))`

Floating-point conversions, e. g. `F64toF32(`rm`,`q`)`	Analogous application to the dot values. `F64toF32(`rm`,`$\dot{q}$`)`
Binary reinterpretation of floating-point representations as integers and vice versa, e. g. `ReinterpI64asF64(`q`)`	Analogous application to the dot values. `ReinterpI64asF64(`$\dot{q}$`)`
SIMD (un)packing, e. g. `64x4toV256(`q_3`,`q_2`,`q_1`,`q_0`)`	Analogous application to the dot values. `64x4toV256(`$\dot{q}_3$`,`$\dot{q}_2$`,`$\dot{q}_1$`,`$\dot{q}_0$`)`
Bitwise logical operations, e. g. `And64(`q_1`,`q_2`)`	Handling according to Section 5.3.1. (Represented by a CCall.)
Integer arithmetic, e. g. `Add64(`q_1`,`q_2`)`	Not relevant for AD. `0x0:I64`
Comparisons, e. g. `CmpF64(`q_1`,`q_2`)`	Not relevant for AD. `0x0:I32`

Table 5.4: Forward-mode monitor commands defined by Derivgrind.

`get` ⟨*addr*⟩ `fget` ⟨*addr*⟩ `lget` ⟨*addr*⟩	Print the dot value of the `binary64`, `binary32`, or 80-bit x87 number at the memory address *addr*.
`set` ⟨*addr*⟩ ⟨*val*⟩ `fset` ⟨*addr*⟩ ⟨*val*⟩ `lset` ⟨*addr*⟩ ⟨*val*⟩	Set the dot value of the `binary64`, `binary32`, or 80-bit x87 number at the memory address *addr* to the specified value *val*.

5.4. Identifying Input and Output Variables

In Sections 5.2 and 5.3, we have described the forward-mode AD instrumentation that propagates the dot values in the shadow storage (Section 5.1) alongside the execution of the client program. The dynamic binary instrumentation tool Derivgrind only operates on VEX code, which the Valgrind framework produces from the machine code of the compiled client program only. Thus, until this point, there is no recourse to any source code of the client program.

However, some knowledge on the internal structure of the client program is required in order to identify AD input and output variables; and this information must be passed to Derivgrind in order to seed or read their dot values. In this section, we describe several mechanisms to specify these variables to Derivgrind.

5.4.1. Variables Specified by Line Number and Name in the Source Code

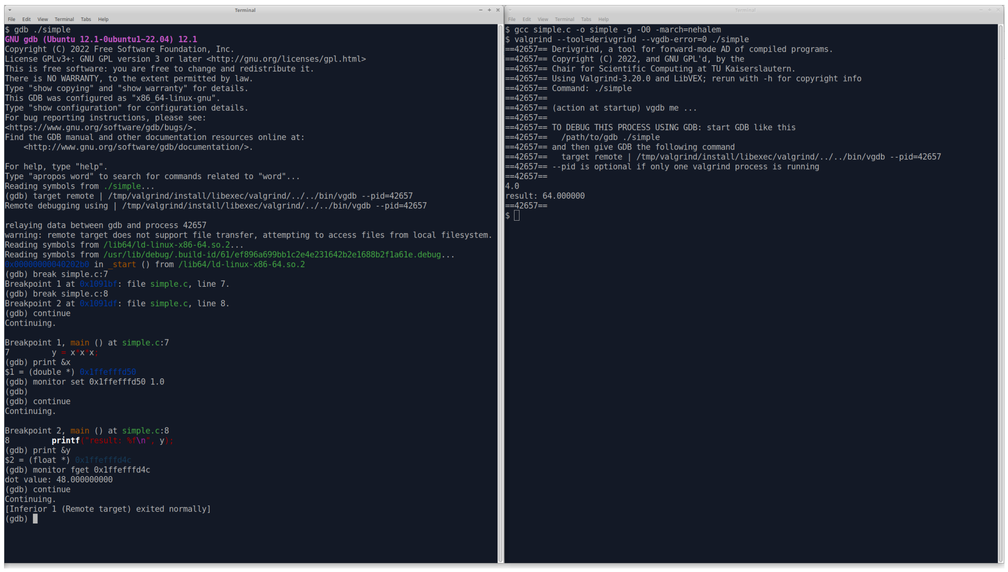
First, let us suppose that the Derivgrind user intends to specify variables by references to file names, line numbers and variable names in the source code of the client program. In this case, Derivgrind naturally needs access to those parts of the source code, or debugging symbols generated from it.

5.4.1.a. Monitor Command Interface. As further detailed in Section 4.3.1, the user can interact with a Valgrind session running an instrumented client program, by interactively typing *monitor commands* into a connected debugger session. Table 5.4 lists the monitor commands defined by Derivgrind to give the user access to the shadow memory, and hence the dot value of any variable. Figure 5.3 displays the result of a Valgrind-GDB session for the client program in Listing 5.1. The C code has been compiled with the GCC flags `-g` and `-O0`, so the debugger can find the appropriate lines and addresses.

5.4.1.b. Client Request Interface. As further detailed in Section 4.3.2, the user can also insert specific instruction sequences into the client program, to invoke *client requests* every time they are executed. Table 5.5 lists the forward-mode client requests provided by Derivgrind; similar to the monitor commands, the first two of them allow to access the shadow memory storing the dot value of any variable in the client program. Listing 5.2 shows how the client requests and an output statement for the derivative are inserted into

Listing 5.1: C program `simple.c`. Figure 5.3 demonstrates how the monitor commands interface can be used to seed $\dot{x}$ and read $\dot{y}$. As an alternative, Listing 5.2 shows insertions that use the client request interface.

```c
#include <stdio.h>

int main(){
  double x;
  float y;
  scanf("%lf", &x);
  y = x*x*x;
  printf("result: %f\n", y);
}
```



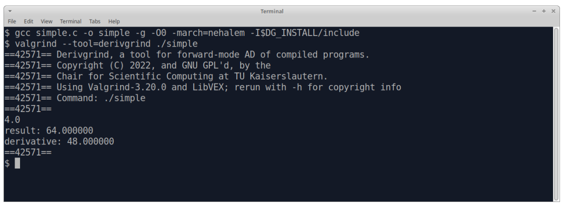
```
$ gdb ./simple
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./simple...
(gdb) target remote | /tmp/valgrind/install/libexec/valgrind/../../bin/vgdb --pid=42657
Remote debugging using | /tmp/valgrind/install/libexec/valgrind/../../bin/vgdb --pid=42657

relaying data between gdb and process 42657
warning: remote target does not support file transfer, attempting to access files from local filesystem.
Reading symbols from /lib64/ld-linux-x86-64.so.2...
Reading symbols from /usr/lib/debug/.build-id/61/ef896a699bb1c2e4e231642b2e1688b2f1a61e.debug...
0x00000000040202b0 in _start () from /lib64/ld-linux-x86-64.so.2
(gdb) break simple.c:7
Breakpoint 1 at 0x1091bf: file simple.c, line 7.
(gdb) break simple.c:8
Breakpoint 2 at 0x1091df: file simple.c, line 8.
(gdb) continue
Continuing.

Breakpoint 1, main () at simple.c:7
7         y = x*x*x;
(gdb) print &x
$1 = (double *) 0x1ffefffd50
(gdb) monitor set 0x1ffefffd50 1.0
(gdb)
(gdb) continue
Continuing.

Breakpoint 2, main () at simple.c:8
8         printf("result: %f\n", y);
(gdb) print &y
$2 = (float *) 0x1ffefffd4c
(gdb) monitor fget 0x1ffefffd4c
dot value: 48.000000000
(gdb) continue
Continuing.
[Inferior 1 (Remote target) exited normally]
(gdb) 
```

```
$ gcc simple.c -o simple -g -O0 -march=nehalem
$ valgrind --tool=derivgrind --vgdb-error=0 ./simple
==42657== Derivgrind, a tool for forward-mode AD of compiled programs.
==42657== Copyright (C) 2022, and GNU GPL'd, by the
==42657== Chair for Scientific Computing at TU Kaiserslautern.
==42657== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==42657== Command: ./simple
==42657==
==42657== (action at startup) vgdb me ...
==42657==
==42657== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==42657==   /path/to/gdb ./simple
==42657== and then give GDB the following command
==42657==   target remote | /tmp/valgrind/install/libexec/valgrind/../../bin/vgdb --pid=42657
==42657== --pid is optional if only one valgrind process is running
==42657==
4.0
result: 64.000000
==42657==
$ 
```

Figure 5.3: GDB session (left) connected to a Derivgrind session (right) differentiating the code in Listing 5.1. The user accesses the dot values $\dot{x}$ and $\dot{y}$ via monitor commands in the GDB session. The client program has been compiled with debugging symbols (`-g`) and optimizations turned off (`-O0`).

Table 5.5: Forward-mode client request macros defined by Derivgrind.

`DG_SET_DOTVALUE(`⟨*valaddr*⟩`,` ⟨*dotvaladdr*⟩`,` ⟨*size*⟩`)`	Invoked with pointers to variables `*`⟨*valaddr*⟩ and `*`⟨*dotvaladdr*⟩ of the same floating-point format, and the size of the format in bytes. Seeds the dot value of `*`⟨*valaddr*⟩ with the present value of `*`⟨*dotvaladdr*⟩. The dot value of `*`⟨*dotvaladdr*⟩ is ignored.
`DG_GET_DOTVALUE(`⟨*valaddr*⟩`,` ⟨*dotvaladdr*⟩`,` ⟨*size*⟩`)`	Same arguments as `DG_SET_DOTVALUE`; copies the dot value of `*`⟨*valaddr*⟩ into `*`⟨*dotvaladdr*⟩, and leaves the dot value of `*`⟨*dotvaladdr*⟩ unspecified.
`DG_GET_MODE`	Evaluates to `'d'` if the client is running under the forward mode of Derivgrind, `'b'` for the reverse mode (Chapter 6) and `'t'` for the bit-trick finding mode (Chapter 7).
`DG_DISABLE(`⟨*plus*⟩`,` ⟨*minus*⟩`)`	Increments an internal thread-local counter `dg_disable[threadID]` of Derivgrind by (⟨*plus*⟩ − ⟨*minus*⟩), and returns the previous value of the counter.



```
$ gcc simple.c -o simple -g -O0 -march=nehalem -I$DG_INSTALL/include
$ valgrind --tool=derivgrind ./simple
==42571== Derivgrind, a tool for forward-mode AD of compiled programs.
==42571== Copyright (C) 2022, and GNU GPL'd, by the
==42571== Chair for Scientific Computing at TU Kaiserslautern.
==42571== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==42571== Command: ./simple
==42571==
4.0
result: 64.000000
derivative: 48.000000
==42571==
$ 
```

Figure 5.4: Derivgrind session differentiating the code in Listing 5.2.

the source code of the client program, and Figure 5.4 shows the output in a Derivgrind session. Additional client requests `DG_GET_MODE` and `DG_DISABLE` have been implemented for internal use in the math wrappers, as outlined in Section 5.5.

5.4.2. Variables Specified as Arguments of Compiled Functions

In contrast to Section 5.4.1, the client code could also be available as a library function with a known signature that comprises all input and output arguments. In this case, the user can apply Derivgrind to a small "library caller" client program stub that seeds the dot values of the input variables, calls the library function, and reads and outputs the dot values of the output variables. This procedure does not require access to the source code of the client code library.

We will use this approach in Section 5.5 to study the black-box derivatives of mathematical functions in the glibc math library, and in Section 6.5.2 to provide an interface between Derivgrind and machine learning frameworks.

Listing 5.2: Insertion of client request macros into the C program `simple.c` of Listing 5.1. When run under Derivgrind, the modified client program accesses the dot values $\dot{\mathtt{x}}$ and $\dot{\mathtt{y}}$ via client requests, and prints $\dot{\mathtt{y}}$ to standard output, as shown in Figure 5.4.

```
  #include <stdio.h>
+ #include <valgrind/derivgrind.h>

  int main(){
    double x;
    float y;
    scanf("%lf", &x);
+   double const x_d = 1.0;
+   DG_SET_DOTVALUE(&x, &x_d, sizeof(double));
    y = x*x*x;
    printf("result: %f\n", y);
+   float y_d;
+   DG_GET_DOTVALUE(&y, &y_d, sizeof(float));
+   printf("derivative: %lf\n", y_d);
  }
```

5.4.3. Variables Accessible by an Add-On Mechanism

The client programs CPython and LibreOffice Calc, to which we will apply Derivgrind in Sections 8.2 and 8.3, feature an “add-on mechanism”. During the run-time of the program, this mechanism enables to user to make the program load and run user-supplied code with (partial) access to its data. The “injected” add-on code can contain client requests, and if the AD input and output variables can be exposed to it, no modification of the source code of the client program is necessary at all.

5.5. Math Function Wrappers

The C standard library provides basic maths functions such as power and square root, the trigonometric and hyperbolic functions and their inverses, exponentiation and logarithm. While some of them could be realized by hardware instructions like `fsin` and `fcos`, implementations of the standard library are free to perform an approximation algorithm entirely in software, and glibc does so.

Figure 5.5 shows the forward-mode automatic derivatives of the glibc version 2.35 math library's implementation of `sin` and `log`, using the components of Derivgrind presented so far. They agree with the analytic derivatives $\cos(x)$ and $1/x$ only inside the intervals $[-0.126, 0.126]$ and $[0.9375, 1.0646972656\dots]$, respectively, and are zero outside. We further analyzed the case of `sin` in glibc version 2.35, with the following findings:

- For $|x| < 2^{-26}$ and $2^{-26} \leq |x| < 0.126$, `sin(x)` is computed using the Taylor polynomials of degree 1 or 11, respectively. Derivgrind thus computes the derivatives of these polynomials, which equal the Taylor polynomials of degree 0 or 10 to the cosine function, and are therefore good approximations for the analytical derivative in the respective intervals.

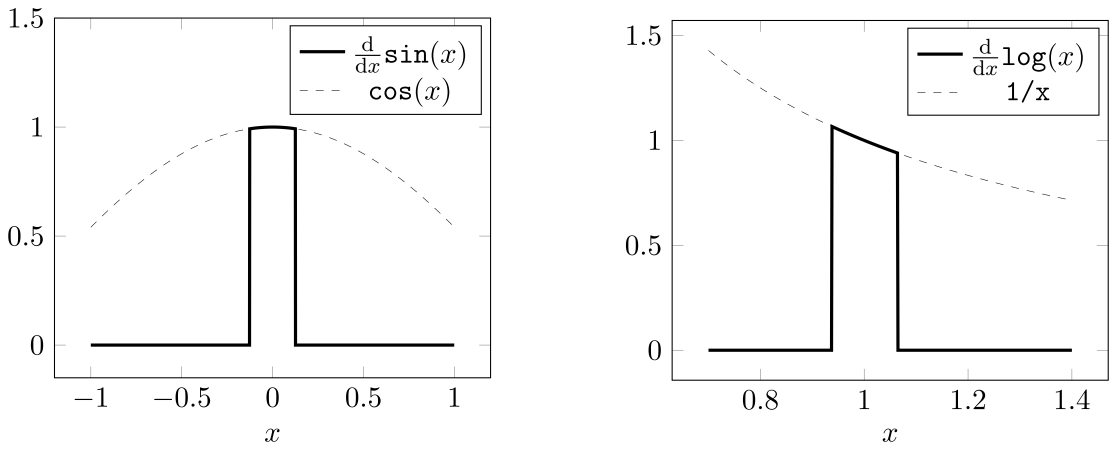


Figure 5.5: The black-box algorithmic derivative of glibc's implementation of `sin` and `log` agrees with the analytic derivatives of the mathematical functions sin, log only in parts of their domains.

- For $0.126 \leq |x| < 0.855\ldots$, the algorithm in glibc is based on a trigonometric formula

$$\sin(x_{\text{tab}} + x_{\text{rem}}) = \sin(x_{\text{tab}})\cos(x_{\text{rem}}) + \cos(x_{\text{tab}})\sin(x_{\text{rem}})$$

  after writing x as $x_{\text{tab}} + x_{\text{rem}}$ with a multiple x_{tab} of 2^{-7} and a small remainder x_{rem}. The purpose of this decomposition is to read the sine and cosine of x_{tab} from a lookup table, and to use a Taylor series for x_{rem}. While the correct decomposition of $\dot{x}$ would be $\dot{x}_{\text{tab}} = 0$ and $\dot{x}_{\text{rem}} = \dot{x}$, Derivgrind erroneously computes $\dot{x}_{\text{tab}} = \dot{x}$ and $\dot{x}_{\text{rem}} = 0$, because glibc performs the decomposition by adding and subtracting a big constant, relying on a bit-trick related to floating-point errors as described in Section 3.3.5. We provide more details and a closer look at the code in Appendix A.3.

- For $0.855\ldots \leq |x| < 2.426\ldots$, the implementation of `cos` is invoked with a modified value, basically using the same lookup-table based approach.

- For $2.426\ldots < |x| < 1.054\ldots \cdot 10^8$, the previously mentioned methods are used for a shifted argument $y = x - k \cdot \frac{\pi}{2} \in [-\frac{\pi}{4}, \frac{\pi}{4}]$. As glibc again computes the integral factor k by a tricky exploitation of floating-point errors as described in Section 3.3.5, Derivgrind erroneously finds $\dot{k} = \dot{x} \cdot \frac{2}{\pi}$ and $\dot{y} = 0$.

To summarize, the glibc 2.35 implementation of `sin` relies on unsupported bit-tricks. This kind of problem should be expected in many functions of a highly performance-optimized math library. We solve it using Valgrind's function wrapping feature (Section 4.3.3) to intercept and redirect calls to many `math.h` functions. In a first version shown in Listing 5.3 and extended in the next chapters, our wrapper

(a) evaluates the original function from `libm.so` to obtain the return value, Listing 5.3: Valgrind function wrapper for the function `sin` from `libm.so*`. Derivgrind's forward mode employs a wrapper of this kind to propagate dot values according to the analytical differentiation rule for sin, instead of a black-box differentiation of the numerical approximation algorithm. The wrapper is further developed in Listings 6.2 and 7.1.

```c
#include "valgrind.h"
#include "derivgrind.h"
#include <math.h>
#include <stdbool.h>

__attribute__((optimize("O0")))
double I_WRAP_SONAME_FNNAME_ZU(libmZdsoZa, sin) (double x) {
  OrigFn fn;
  VALGRIND_GET_ORIG_FN(fn);
  bool already_disabled = DG_DISABLE(1,0)!=0;
  double ret;
  CALL_FN_D_D(ret, fn, x);                    /* ←(a) */
  double ret_d = ret;
  if(!already_disabled) {
    if(DG_GET_MODE=='d'){ /* forward mode */
      double x_d;
      DG_GET_DOTVALUE(&x, &x_d, 8);           /* ←(b) */
      double ret_d = (cos(x)) * x_d;          /* ←(c) */
      DG_SET_DOTVALUE(&ret, &ret_d, 8);       /* ←(d) */
      DG_DISABLE(0,1);
    } /* will add code for the recording pass and bit-trick finder ↩
         modes */
  } else {
    DG_DISABLE(0,1);
  }
  return ret;
}

/* ... wrappers for other math.h functions ... */
```

(b) uses a client request to obtain the dot value of the argument,

(c) evaluates the partial derivative and uses it to compute the dot value of the return value according to (2.15), and

(d) uses a client request to set the dot value of the return value.

The macro `CALL_FN_D_D` allows to call the original function in `libm.so` with the signature `double sin(double)` in step (a). We have created this macro and added it to Derivgrind's version of `valgrind.h`, adapting the analogous macro `CALL_FN_W_W` for functions with integer arguments and return values; see Appendix B.2 for more details.

The call to the original `libm.so` in step (a) is instrumented by Derivgrind, but the resulting dot value in `ret`, which might have been subject to unsupported bit-tricks, is overwritten by the client request in step (d). If the partial derivatives in step (c) involve math functions, Valgrind redirects them again to their wrappers. Without further care, the `sin` wrapper would call the `cos` wrapper, which would in turn call the `sin` wrapper, which would call the cos wrapper again, and so on, leading to an infinite recursion. To prevent this, Derivgrind keeps a thread-local internal integer dg_disable[threadID]. The counter is supposed to be zero most of the time while the client program is running. When the client program calls a wrapped math function, the first DG_DISABLE client request in Listing 5.3 increments the counter by 1 and returns zero (according to Table 5.5), so already_disabled is false and the wrapper will evaluate the partial derivative. If this involves another call to a wrapped math function, such as to cos in the wrapper for sin, the first DG_DISABLE request of this nested call returns 1, so it does not make another math library call itself. Each call to a math function will lead to a second DG_DISABLE request in the end to decrement the counter by 1.

On the implementation side, we use a Python script to generate function wrapper code like Listing 5.3 for most math.h functions.

The function wrapping approach comes with limitations. When compiler optimizations are turned on, GCC sometimes "inlines" a bit-trick instead of inserting a call to the math library, as shown in Listings 3.17 and 3.21. Naturally, the function wrapping mechanism does not become active in this case. Likewise, if a client program implements numerical approximations of mathematical functions on its own and uses different function names or inlining, AD tools based on machine code can hardly recognize these, and thus fall back to black-box differentiation. We will see examples of this in Paragraph 8.2.d and Appendix A.2 for binary32 math functions in the NumPy library, and in Appendix A.1 with the G4Log function in the particle simulation framework Geant4.

And the other way round, if a client program reuses math.h function names in a shared object libm.so* with a different semantic or signature, the function wrapping mechanism will still assume the math.h semantic and signature, which might lead to unexpected behaviour.

6. Reverse-Mode Automatic Differentiation of Compiled Programs

In the previous Chapter 5, we have presented Derivgrind as a forward-mode AD tool applicable to compiled programs. Implemented in the Valgrind framework for dynamic binary instrumentation, Derivgrind instruments VEX code representing the client program with AD logic. Specifically, Derivgrind's forward-mode AD logic keeps track of dot values for all floating-point numbers handled by the client program. Derivgrind's monitor commands and client requests, as described so far, provide access to these dot values to the interactive user and the client program, respectively. Math wrappers use analytic differentiation rules to properly propagate dot values through functions in the C math library despite its many bit-tricks.

In this chapter, we describe another set of instrumentation, monitor commands, client requests and math wrappers available in Derivgrind, suitable for reverse-mode AD. More specifically, Derivgrind is capable of performing a tape-recording forward pass (Section 2.4.5) with linear index management and Jacobian taping while the compiled client program executes, and a separate tape evaluator program will be used for the reverse pass. There is a lot of structural similarity to the previous Chapter 5, and as the instrumentation is much more complex, we will take a slightly higher-level perspective in this chapter.

Specifically, Derivgrind's forward-pass AD logic assigns and keeps track of indices for all floating-point numbers handled by the client program, and records indices and partial derivatives for the relevant real-arithmetic steps performed by the client program. We specify the way of storing indices in Section 6.1, and the tape layout in Section 6.2; these specifications are implemented by the instrumentation described in Sections 6.3 and 6.4. Section 6.5 lists additional monitor commands and client requests that give access to these identifiers, and Section 6.6 extends the math wrappers of Derivgrind to record analytic derivatives for functions in the C math library.

After the recording has finished, the tape file can be used by a separate *tape evaluator* program, described in Section 6.7, to compute reverse-mode automatic derivatives. Additionally, the tape evaluator offers an alternative way to compute forward-mode automatic derivatives using the procedure in Paragraph 2.4.5.c. This is why we avoid the term "reverse-mode instrumentation" in the remainder of this chapter, and prefer the more precise terms *forward/recording pass* or *index-handling and tape-recording* instrumentation.

6.1. Shadow Data: Identifiers

Derivgrind's index-handling and tape-recording instrumentation assigns a 64-bit integer *index* $\hat{a}$ to each real number a appearing in the client program.

6.1.a. Choice of Index Management. The default index of passive numbers is 0. Active numbers are consecutively assigned the indices 1, 2, 3, ..., in the order in which they are declared as AD inputs or computed by real-arithmetic statements. To cover some corner case details,

- the assigned indices shall increase monotonically and unnecessarily large gaps should be avoided, but it is acceptable to reserve identifiers for internal use by the AD tool;
- if several scalar arithmetic operations are applied at the same time, e. g. by SIMD operations, the order in which indices are assigned to the scalar components of the result is unspecified; and
- we allow mere copies of a floating-point number to share its index; this makes the AD instrumentation of data moves very easy.

In the nomenclature of Paragraph 2.4.5.a, this approach is known as *linear index management* with *copy optimization*[159].

6.1.b. Shadowing Policy. Similar to the dot value in the forward mode (Section 5.1), the index $\hat{a}$ of a floating-point number a is held in shadow storage locations of the same size and type as the temporary, register, or memory location storing a. For the recording pass however, two layers of these shadow locations are required, because we need to be able to store a 64-bit index even if the floating-point number occupies only 32 bit. They are named *upper* and *lower* layer, and used in the following way.

Whenever a storage location contains the entire binary representation of a floating-point value a in any floating-point format, *which occupies at least four bytes*, Derivgrind's index handling instrumentation shall make sure that the lowest four bytes in the upper and lower shadow locations store the upper and lower 32-bit halves $\hat{a}^{\texttt{high}}$ and $\hat{a}^{\texttt{low}}$ of the index $\hat{a}$, respectively. Beyond the lowest four bytes, the content of the shadow location is unspecified. This is illustrated in Figure 6.1.

This rule shall generalize to parts of floating-point representations (e. g., individual bytes extracted from a `binary64`), and to storage locations only partially filled with such (e. g., a 256-bit SIMD vector containing a `binary32` and 224 unused bits), in the obvious way: If a byte belongs to a floating-point representation of a number a, the corresponding bytes in the upper and lower shadow locations shall store the respective bytes of $\hat{a}^{\texttt{high}}$ and $\hat{a}^{\texttt{low}}$, respectively. This is illustrated in Figure 6.2.

When bytes in a memory location do not originate from, and are not further used as floating-point data, the corresponding bytes in the upper and lower shadow locations are unspecified.

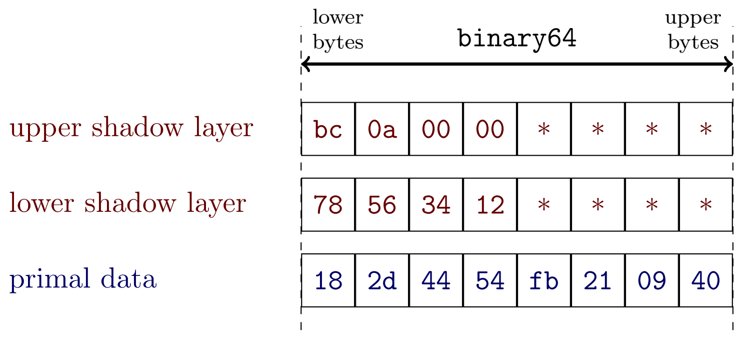


Figure 6.1: The 64-bit index `0x00000abc12345678` is stored in the lowest four bytes of the two shadow locations (red) over the `binary64` number `0x400921fb54442d18` (blue, representing the value $\pi \approx 3.14$). The remaining four bytes of either shadow memory location are unspecified (indicated by red stars).

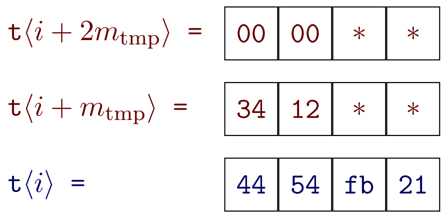

$\mathtt{t}\langle i + 2m_{\mathrm{tmp}}\rangle$ =
00 00 * *
$\mathtt{t}\langle i + m_{\mathrm{tmp}}\rangle$ =
34 12 * *
$\mathtt{t}\langle i\rangle$ =
44 54 fb 21

Figure 6.2: Suppose that the four-byte temporary $\mathtt{t}\langle i\rangle$ is assigned with the middle four bytes of the `binary64` variable in Figure 6.1. According to Paragraph 6.1.b, the two shadow temporaries (red) store the respective parts of the shadow data in Figure 6.1.

In particular, the default content of uninitialized shadow memory should be zero, so global variables and constants are automatically considered to be passive variables at first.

6.1.c. Rationale. We arrived at this exact form of the policy from the following considerations:

- We cannot follow a *reuse index management* approach (Paragraph 2.4.5.a) because at the moment, we have no robust and efficient means to recognize when the last copy of a variable disappears.
- As a consequence, we cannot guarantee that a 32-bit index space is sufficient: Roughly estimating that the client program performs one elementary real-arithmetic step per CPU clock cycle and the clock frequency is 1 GHz, the index would overflow $2^{32} - 1 \approx 4.29 \cdot 10^9$ after recording about four seconds' worth of primal run-time.
- However, a 48-bit index space should suffice. As discussed below in Section 6.2, at least 16 B of tape space are allocated for every index, so if there was an overflow of

a 48-bit unsigned integer, the tape would already occupy at least 4096 TB, likely overflowing realistic storage systems. Yet, we decided to use 64-bit indices to simplify the implementation.

- The provisions for incomplete parts of binary representations, which might make up only part of a memory location, have the same background as in the forward mode (Paragraph 5.1.b): Binary representations may be temporarily splitted e. g. for copy operations, and may fill only a component of a SIMD vector.
- One could treat the 64- and 80-bit floating-point types differently from the 32-bit type, filling the entire 64-bit index into one of the shadow layers instead of splitting it into halves. Also, four-byte index halves might as well be aligned differently, e. g. to the upper end of the floating-point number. All of these alternatives have no obvious advantage.

6.1.d. Summary of the Instrumentation. The index-handling and tape-recording AD logic, implemented in Sections 6.3 and 6.4, updates the shadow locations according to Paragraphs 6.1.a and 6.1.b, and records a tape according to the format described in Section 6.2. The instrumentation is based on the following two main facts:

- For any source and target locations of a data transfer operation, the respective two layers of shadow storage contain the entire AD information about the stored data, in a universal and translation-invariant format. Thus, the inserted AD logic simply moves the content of both shadow layers in exactly the same way as the original data. This type-agnostic rule is analogous to the forward mode (Paragraph 5.1.d).
- If a real-arithmetic operation like min or max returns a copy of one of its operands, the previous rule applies. If, otherwise, a new real number is computed, the instrumentation should assign a new index, and record its relation to the indices of the operands on the tape. Details on the layout of the tape are shared in the next Section 6.2.

6.2. Layout of the Tape

6.2.a. Tape Layout. Derivgrind's index-handling and tape-recording logic produces a *Jacobian tape* (see Paragraph 2.4.5.b). The tape is organized as a sequence of 32-byte blocks, each of which is made up of two 64-bit unsigned integers followed by two `binary64`s. The idea is that the i-th block stores all the necessary information about the statement (2.14) in which the number with index i was computed: The integers store the indices of up to two operands on the right-hand side, and the `binary64`s store the respective partial derivatives. Formally, the following policy applies.

- If the index i was assigned to an AD input or is the default index `0`, the i-th block stores two 8-byte indices `0` and two `binary64`s `0.0`.

- If the index i was assigned to the result a_{lhs} of a binary real-arithmetic step $a_{\mathrm{lhs}} = \phi(a_1, a_2)$, the i-th block stores the 8-byte indices $\hat{a}_1$ and $\hat{a}_2$, as well as the partial derivatives $\frac{\partial\phi}{\partial a_1}(a_1, a_2)$, $\frac{\partial\phi}{\partial a_2}(a_1, a_2)$ as 8-byte `binary64`s.

- If the index i was assigned to the result a_{lhs} of an unary real-arithmetic step $a_{\mathrm{lhs}} = \phi(a_1)$, the i-th block stores the 8-byte index $\hat{a}_1$ along with the 8-byte index `0`, and the partial derivative $\frac{\partial\phi}{\partial_1}(a_1)$ along with the partial derivative `0.0` as `binary64`s.

- If the index i was assigned to the result a_{lhs} of a ternary real-arithmetic step $a_{\mathrm{lhs}} = \phi(a_1, a_2, a_3)$, Derivgrind's index handling logic makes sure that the index $(i-1)$ is reserved for internal use. The $(i-1)$-th block stores the indices $\hat{a}_1$ and $\hat{a}_2$ and the partial derivatives $\frac{\partial\phi}{\partial a_1}(a_1, a_2, a_3)$ and $\frac{\partial\phi}{\partial a_2}(a_1, a_2, a_3)$. The i-th block stores the indices $\hat{a}_3$ and $(i-1)$ and the partial derivatives $\frac{\partial\phi}{\partial a_3}(a_1, a_2, a_3)$ and `1.0`.

- In VEX, there are no real-arithmetic steps with more than three operands relevant for AD.

- If the index i was otherwise internally used by the index-handling and tape-recording logic, the content of the i-th block is unspecified.

Figure 6.3 displays the state of the tape and the index files after two variables $x_1 = 3$, $x_2 = -4$ were declared as AD inputs, then multiplied, and the result was declared as an AD output.

6.2.b. Rationale. The above tape layout was selected due to the following considerations.

- In our setup, Jacobian taping is much easier to implement than primal value taping (Paragraph 2.4.5.b), which requires more complex data structures to record handles that indicate operations, passive constants, etc.

- As discussed in Paragraph 6.1.c, the set of 48-bit integers is large enough to eliminate the risk of index overflows in any setup with a realistic tape size, because the partial derivatives in each block occupy at least 16 B. Even though the index handling logic uses 64-bit indices, it would be possible to record only the least-significant 48 bit on the tape. Such a change of the above layout would reduce the tape size of each block from 32 B to 28 B (saving 12.5 %), at the cost of a more complex implementation.

- Instead of fixing the number of index-derivative pairs per block to two, one could start a block with the two-bit information whether zero, one, two or three of these pairs follow. These two bits could be placed in the most-significant 16 bit of the first index (i. e. those beyond the previously mentioned 48 bit) of each block, with a special arrangement for blocks with zero pairs. Again, such a change would moderately reduce the tape size while making the tape layout much more complex.

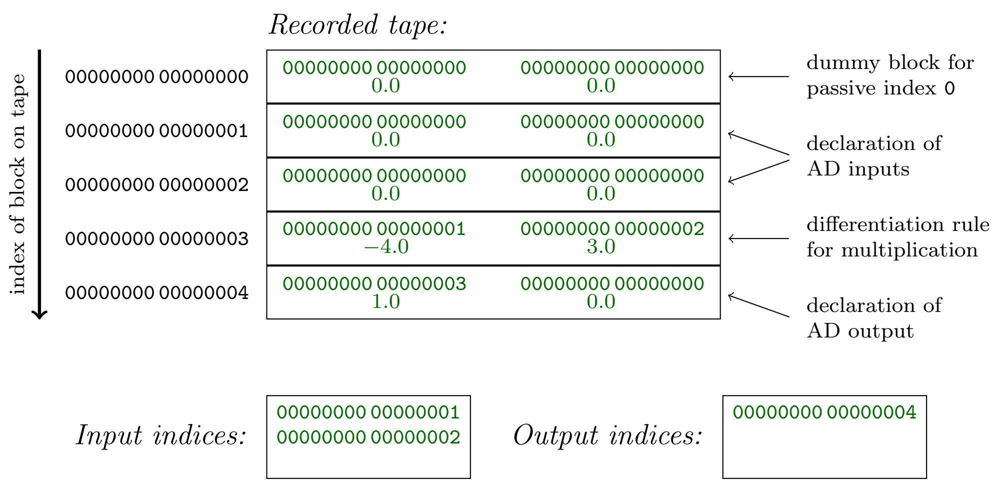


Figure 6.3: Tape and index files after declaring two variables $x_1 = 3$, $x_2 = -4$ as AD inputs, multiplying them, and declaring the result $y = x_1 \cdot x_2$ as an AD output. Each of the five tape blocks stores two indices (as 8-byte unsigned integers) and two partial derivatives (as 8-byte `binary64`s, denoted here as decimal floating-point numbers).

6.3. Instrumentation of Statements

6.3.a. Recap from the Forward Mode. Implemented in the Valgrind framework for dynamic binary instrumentation, Derivgrind inserts the AD logic into VEX superblocks presented by the Valgrind core. In the forward mode (Chapter 5), statements with relevance to AD were prepended with dot-value-propagating statements, as outlined in Section 5.2 and Table 5.1. Basically, dot-value-propagating statements copy shadow data where the primal statement copies data, and may involve dot-value-propagating expressions $\dot{p}$ for expressions p in the primal statement. The expression $\dot{p}$ is formed according to analytic differentiation rules as detailed in Section 5.3 and Tables 5.2 and 5.3, so when p evaluates to floating-point data, $\dot{p}$ evaluates to the respective dot values.

6.3.b. Recording-Pass Instrumentation. Analogous to the forward mode, additional *index-handling and tape-recording statements* are inserted in front of each primal VEX statement (except for CAS statements, see Paragraph 6.3.c). Basically, index-handling and tape-recording statements copy indices whenever the primal statement copies a value, as shown in Table 6.1. When a primal statement contains an expression p, the index-handling and tape-recording statement can involve *index-handling and tape-recording expressions* $\hat{p}^{\texttt{low}}$, $\hat{p}^{\texttt{high}}$. These evaluate to the content of the lower and upper shadow layers associated with the value of p, respectively, according to the shadowing policy from Paragraph 6.1.b. We discuss the construction of $\hat{p}^{\texttt{low}}$ and $\hat{p}^{\texttt{high}}$ in the next Section 6.4.

Table 6.1: Augmentation of VEX statements with index-handling and tape-recording AD logic. The structure of differentiated statements is largely analogous to the forward mode (Table 5.1), just with two shadow layers instead of one.

VEX statement	Additional VEX statements for the recording pass
Store p in a temporary with index i. `t`$\langle i\rangle$ `=` p	Store $\hat{p}$ in the shadow temporaries. `t`$\langle i+m_{\text{tmp}}\rangle$ `=` $\hat{p}^{\texttt{low}}$ `t`$\langle i+2m_{\text{tmp}}\rangle$ `=` $\hat{p}^{\texttt{high}}$
Store p in the register with byte offset j in the guest state. `PUT(`$\langle j\rangle$`)` `=` p	Store $\hat{p}$ in the shadow registers. `PUT(`$\langle j+m_{\text{gs}}\rangle$`)` `=` $\hat{p}^{\texttt{low}}$ `PUT(`$\langle j+2m_{\text{gs}}\rangle$`)` `=` $\hat{p}^{\texttt{high}}$
Store p in memory. `STle(`*address*`)` `=` p	Store $\hat{p}$ in shadow memory. Implemented as a dirty call to access the shadow memory from VEX.
Store p in memory, if a condition is satisfied. `if (`*guard*`) STle(`*address*`)` `=` p	Store $\hat{p}$ in shadow memory, subject to the same condition. Implemented as a dirty call.
Compare-and-swap, loading *addr* into temporary `t`$\langle$*old*$\rangle$, and replacing data at *addr* by *new* if it matches *expd*. `t`$\langle$*old*$\rangle$ `= CASle(`*addr* `::` *expd* `->` *new*`)`	CAS statements are replaced as discussed in Paragraph 6.3.c.
Dirty call, invoking a Valgrind function with side effects.	The augmentation depends on the dirty call, see details in Paragraph 6.3.d.
Meta information. `------ IMark(...) ------`	Not relevant for AD.
Conditional jump. `if (`*guard*`) goto {`$\langle$*jump kind*$\rangle$`}` $\langle$*target*$\rangle$	Not relevant for AD.

6.3.c. Compare-And-Swap Statements. Paragraph 6.3.b comes with the same exception for CAS statements as in the forward mode (Section 5.2.2): They are replaced by an equivalent (but non-atomic) sequence of copy operations depending on a check whether a shared state has been modified, and this shared state includes a few bytes in memory as well as the respective bytes in either layer of the shadow memory. For the same reason as in the forward mode (Section 5.2.2), this is not sufficient to support multi-threading.

6.3.d. Dirty Call Statements. As Derivgrind's representation of an index $\hat{a}$ is independent of the floating-point format used to store a (Paragraph 6.1.b), the dirty calls converting between `binary64` and the 80-bit x87 floating-point type are instrumented by mere copy operations on the respective shadow data. As in the forward mode, the other dirty calls (see Table 4.3) are not instrumented except that the shadows of their output temporaries are zeroed.

6.4. Instrumentation of Expressions

6.4.a. Data Transfer Expressions. Many of the index-handling and tape-recording statements in Table 6.1 and Paragraphs 6.3.c and 6.3.d involve index-handling and tape-recording expressions $\hat{p}^{\texttt{low}}$, $\hat{p}^{\texttt{high}}$, which represent the 32-bit halves of the 64-bit index assigned to the result of an expression p in the original statement. Table 6.2 shows how Derivgrind constructs $\hat{p}^{\texttt{low}}$, $\hat{p}^{\texttt{high}}$ for expressions that basically transfer data; the shadow data is moved alongside, similar to the forward mode.

6.4.b. Operations. Concerning expressions that perform operations (Tables 4.5 and 4.6), Derivgrind's recording-pass instrumentation handles precisely the same set of operations as the forward-mode instrumentation. However, the index-handling and tape-recording instrumentation is much more complex than the dot-value-propagating instrumentation in Table 5.3: While computing the dot value does not introduce side effects and is therefore implemented in VEX by a pure dot-value-propagating expression, pushing a block to the tape is a side effect and therefore requires one or multiple separate VEX statements to be added in front of the primal and index-handling and tape-recording statements.

6.4.c. Operation Handling Example: `Div32Fx8`. As an generic example, we illustrate the index-handling and tape-recording instrumentation of the operation `Div32Fx8(`q`, `s`)` with expressions q, s. This expression evaluates to the componentwise floating-point quotient in all eight `binary32` components of the `V256` operands that q and s evaluate to.

For each component $i = 0, \ldots, 7$, Derivgrind's recording-pass expression handling will insert a dirty call statement into the instrumented VEX superblock. The dirty call takes six expressions as arguments, four of which extract the components

$$\hat{q}_i^{\texttt{low}}, \hat{q}_i^{\texttt{high}}, \hat{s}_i^{\texttt{low}}, \hat{s}_i^{\texttt{high}} \tag{6.1}$$

Table 6.2: Construction of index-handling and tape-recording expressions.

Expression p	Index-handling and tape-recording expressions $\hat{p}^{\texttt{low}}$, $\hat{p}^{\texttt{high}}$
Read from a temporary with index i. `t`$\langle i \rangle$	Read from the shadow temporary. `t`$\langle i + m_{\text{tmp}} \rangle$, `t`$\langle i + 2m_{\text{tmp}} \rangle$
Read data of specified type from the register with byte offset j in the guest state. `GET:`$\langle type \rangle$`(`$\langle j \rangle$`)`	Read data of the same type from the shadow register. `GET:`$\langle type \rangle$`(`$\langle j + m_{\text{gs}} \rangle$`)`, `GET:`$\langle type \rangle$`(`$\langle j + 2m_{\text{gs}} \rangle$`)`
Read from memory. `LDle:`$\langle type \rangle$`(`*address*`)`	Read from shadow memory, expecting data of the same type. Implemented as dirty calls.
Operation with one to four arguments, see Table 5.3 for examples. $\langle op \rangle$`(`q_1`, `q_2`, ... )`	See Paragraph 6.4.c.
Constant value. $\langle literal \rangle$`:`$\langle type \rangle$ or $\langle type \rangle$`{`$\langle literal \rangle$`}`	Constant for the default index zero of the same type. `0x0:`$\langle type \rangle$ or $\langle type \rangle$`{0x0}`
If-then-else construct, selecting either a or b depending on *condition*. `ITE(`*condition*`, `q_1`, `q_2`)`	If-then-else construct with same condition on differentiated operands. `ITE(`*condition*`, `$\hat{q}_1^{\texttt{low}}$`, `$\hat{q}_2^{\texttt{low}}$`)`, `ITE(`*condition*`, `$\hat{q}_1^{\texttt{high}}$`, `$\hat{q}_2^{\texttt{high}}$`)`
CCall to Valgrind function without side effects.	Until now, we only encountered cases without relevance to AD.

from $\hat{q}^{\texttt{low}}, \hat{q}^{\texttt{high}}, \hat{s}^{\texttt{low}}, \hat{s}^{\texttt{high}}$ (and pad them to `I64`s). The other two arguments to the dirty call are `I64`-reinterpretations of the partial derivatives

$$\texttt{DivF64(}rm\texttt{, F64\{1.0\}, }s'_i\texttt{)} \tag{6.2}$$

and

$$\texttt{DivF64(}rm\texttt{, MulF64(}rm\texttt{, F64\{-1.0\}, }q'_i\texttt{), MulF64(}rm\texttt{, }s'_i\texttt{, }s'_i\texttt{))}, \tag{6.3}$$

of $\texttt{DivF64(}rm\texttt{,}q'_i\texttt{,}s'_i\texttt{)}$ with respect to $q'_i = \texttt{F32toF64(}q_i\texttt{)}$ and $s'_i = \texttt{F32toF64(}s_i\texttt{)}$.

The dirty call passes the values of these six expressions to the C function

```
dg_bar_writeToTape_call(ULong index1Lo, ULong index1Hi,
            ULong index2Lo, ULong index2Hi, ULong diff1, ULong diff2)
```

which is part of the Derivgrind tool. It assembles two 64-bit indices from $\hat{q}_i^{\texttt{low}}, \hat{q}_i^{\texttt{high}}$ and $\hat{s}_i^{\texttt{low}}, \hat{s}_i^{\texttt{high}}$, and reinterprets the partial derivatives as `doubles`. If the index of at least one of the two operands is non-zero, the tape block is written to a buffer. When the tape buffer is full, the function flushes it into the binary tape file, using Valgrind's C standard library replacement functions. The position of the new block on the tape defines the index to be returned.

If the indices of both operands are zero, i. e. there is no active operand, no block is stored and the index zero is returned (activity analysis). The same goes if the thread-local counter `dg_disable[threadID]` (see Section 5.5) is non-zero, for a reason we explain further down in Section 6.6.

Either way, the return value (a new active index or zero) is placed in a yet unused `I64` temporary, behind the temporaries used by the VEX block before instrumentation and their shadows (Paragraph 4.4.b). The index-handling and tape-recording expressions $\hat{p}^{\texttt{low}}$ and $\hat{p}^{\texttt{high}}$ are then assembled from the respective 32-bit halves of the values of these eight temporaries (accessed by VEX temporary read expressions $\texttt{t}\langle\ldots\rangle$).

6.4.d. Operation Handling: Implementation Aspects. The C code to form dot-value-propagating, as well as index-handling and tape-recording VEX expressions, is basically a list of `case` labels and statements for the various kinds of operations in a large `switch` block. There is a lot of repetition in the C code, but also certain details of individual operations needed to be considered, e. g. related to the number of arguments, whether there is a rounding-mode argument, which SIMD combinations (scalar type, vector size, lowest-lane-only) exist etc. We have found it most convenient to generate the C code with a Python script.

6.5. Identifying Input and Output Variables

6.5.1. Monitor Commands and Client Requests

As in the forward mode, Derivgrind defines monitor commands and client requests to allow the user to define AD inputs and outputs for the recording pass.

Table 6.3: Recording-pass monitor commands defined by Derivgrind.

`index` $\langle addr \rangle$	Print the index of the `binary64`, `binary32`, or 80-bit x87 number at the memory address *addr*.
`mark` $\langle addr \rangle$	Assign a new index to the `binary64`, `binary32`, or 80-bit x87 number at the memory address *addr*.

6.5.1.a. Monitor Commands. Derivgrind's recording-pass monitor commands are listed in Table 6.3. As the storage format of an index $\hat{a}$ is independent from the floating-point format used to represent a real number a, the monitor commands `index` and `mark` can be used for `binary64`, `binary32`, and 80-bit x87 numbers in the same way.

6.5.1.b. Client Requests. Derivgrind's reverse-pass client requests are listed in Table 6.4. Additionally, to make declarations of variables as AD inputs and outputs in the client program very easy, we defined combined macros `DG_INPUTF` and `DG_OUTPUTF` as follows.

The macro `DG_INPUTF` expands to a sequence of three of the client requests of Table 6.4 that

1. acquire a new index by pushing a zero block to the tape,
2. initialize the index of the declared AD input variable with this new index, and
3. store the new index in a text file

`dg-input-indices`.

The macro `DG_OUTPUTF` expands to a seqence of three client requests as well, which

1. read the index $\hat{a}$ of the variable declared as AD output,
2. push a block to the tape with indices $\hat{a}$ and `0`, and partial derivatives `1.0` and `0.0`, and
3. store the index corresponding to this block in a text file

`dg-output-indices`.

Creating the additional block for a declared output variable, which represents a copy of it, is not necessary from a technical point of view. However, it ensures that even if the same number is declared as an AD output multiple times, the indices pushed to the output index file are all distinct, helping to prevent bugs in tape evaluation procedures. We colloquially refer to the combined macros `DG_INPUTF` and `DG_OUTPUTF` as client requests, too.

Apart from helping the user to declare AD inputs and outputs, the client requests are also used internally in the Derivgrind package: by the math function wrappers in Section 6.6, by add-ons to CPython and LibreOffice Calc in Sections 8.2 and 8.3, and by the interface to ML frameworks described in Section 6.5.2.

Table 6.4: Additional client request macros defined by Derivgrind for the recording pass. See Table 5.5 for the forward-mode macros.

`DG_GET_INDEX(`⟨*addr*⟩`,` ⟨*iaddr*⟩`)`	Copies the index of the `binary64`, `binary32`, or 80-bit x87 number at *addr* to *iaddr*.
`DG_SET_INDEX(`⟨*addr*⟩`,` ⟨*iaddr*⟩`)`	Sets the index of the `binary64`, `binary32`, or 80-bit x87 number at *addr* according to *iaddr*.
`DG_NEW_INDEX_NOACTIVITYANALYSIS(`⟨*i1addr*⟩`,` ⟨*i2addr*⟩`,` ⟨*d1addr*⟩`,` ⟨*d2addr*⟩`,` ⟨*iaddr*⟩`,` ⟨*valaddr*⟩`)`	Pushes a new block on the tape with the 64-bit unsigned integer indices at *i1addr* and *i2addr* and the `binary64` partial derivatives at *d1addr* and *d2addr*. The index corresponding to the block is stored at *iaddr*. The value of the result can be passed at *valaddr*, but is irrelevant here.
`DG_NEW_INDEX(`⟨*i1addr*⟩`,` ⟨*i2addr*⟩`,` ⟨*d1addr*⟩`,` ⟨*d2addr*⟩`,` ⟨*iaddr*⟩`,` ⟨*valaddr*⟩`)`	Like `DG_NEW_INDEX_NOACTIVITYANALYSIS`, but if the indices at *i1addr* and *i2addr* are both zero, no block is recorded and a zero index is written to *iaddr*.
`DG_INDEX_TO_FILE(`⟨*fileid*⟩`,` ⟨*iaddr*⟩`)`	Append the index at *iaddr* to the file of input or output indices, depending on *fileid*.
`DG_DISABLE(`⟨*plus*⟩`,` ⟨*minus*⟩`)` *(extending the entry of Table 5.5)*	Increments an internal thread-local counter `dg_disable[threadID]` of Derivgrind by (⟨*plus*⟩ − ⟨*minus*⟩), and returns the previous value of the counter. *If the counter is non-zero, no blocks will be recorded on the tape.*

Listing 6.1: External function wrappers exposing Derivgrind-differentiated compiled functions in shared objects to the ML frameworks PyTorch and TensorFlow.

```c
// compilation: gcc thisfile.c -shared -fPIC -o mylib.so -O3
void myfun(int param_size, char* param_buf,
           int input_count, double* input_buf,
           int output_count, double* output_buf) {
  output_buf[0] = 3.14*input_buf[0]+5.0+input_buf[1]*input_buf[2];
}
```

```python
import torch
import derivgrind_torch as dg_torch
x = torch.tensor([4.0,-2.0,6.5],dtype=torch.float64, \
                  requires_grad=True)
y = dg_torch.derivgrind("mylib.so","myfun").apply(b"",x,1)
y.backward()
print(*x.grad.numpy()) # -> 3.14 6.5 -2.0
```

```python
import tensorflow as tf
import derivgrind_tensorflow as dg_tf
x = tf.Variable([4.0,-2.0,6.5],dtype=tf.float64)
with tf.GradientTape() as tape:
  y = dg_tf.derivgrind("mylib.so","myfun").apply(b"",x,1)
dy_dx = tape.gradient(y,x)
print(*dy_dx.numpy()) # -> 3.14 6.5 -2.0
```

6.5.2. Interface to Machine Learning Frameworks

Similar to Enzyme[128], Derivgrind provides an "external function interface" to the ML frameworks PyTorch[146] and TensorFlow[1] for the following use case. Suppose that the user needs to include a custom compiled function in a ML model definition (e. g., as a layer in a deep neural network) without reimplementing the function in the respective ML framework.

To this end, the AD tools underlying these frameworks allow to temporarily pause the recording of their tape, to evaluate an *external function* outside of the ML framework, and to specify *custom derivatives* of this external function in terms of additional external code that performs the back-propagation (2.17). Derivgrind can be used to automatically provide the custom derivative for compiled external functions.

Listing 6.1 illustrates the basic usage of our Python modules `derivgrind_torch` and `derivgrind_tensorflow`. In the first block, a simple external function is defined in the C language, with a specific signature to pass three buffers for non-differentiable parameters (e. g. hyperparameters), AD inputs (e. g. information from the previous layer plus learnable parameters of the external function), and AD outputs (e. g. information for the next layer). The body of this function could call any cross-language or partially closed-source code.

The Python function `apply` in either Python module starts a subprocess to run Derivgrind on the simple library caller program in Appendix B.1, and caches the resulting tape and index files. PyTorch's `Tensor.backward` and TensorFlow's `GradientTape.gradient` functions then trigger a subprocess running the tape evaluator (see Section 6.7) on these files to provide the custom gradient.

6.6. Math Function Wrappers

As described in Section 5.5, black-box differentiation through the numerical approximation algorithms of the glibc math library implementation often leads to incorrect derivatives in the forward mode because of bit-tricks used by these algorithms, but this can be fixed by Valgrind's function wrapping mechanism (Section 4.3.3). In the recording pass, Derivgrind intercepts calls to many `math.h` functions in order to record a tape block with accurate partial derivatives computed by the respective analytic differentiation rules. An example for the `sin` function is shown in Listing 6.2. It has a similar structure as Listing 5.3, evaluating a handle `fn` to the original `sin` function first. Again, `DG_DISABLE` requests are in place to prevent an infinite recursion. Additionally, a non-zero value of `dg_disable[threadID]` disables the tape recording, so we do not store unnecessary tape blocks for real-arithmetic operations performed by the math library implementation. The partial derivative is pushed to the tape with a client request. Another client request assigns the resulting new index to the return value.

6.7. Tape Evaluation

Once the primal program running under Derivgrind, and subsequently the Valgrind process (4.1) itself, have finished, the recording pass is over. The tape and index files

`dg-tape`, `dg-input-indices`, `dg-output-indices` (6.4)

are now ready for reverse- and/or forward-mode evaluation passes. To this end, the Derivgrind package contains a separate program `tape-evaluation`.

6.7.1. Reverse-Mode Tape Evaluation

To specify the bar values of the AD outputs, the user has to store their textual representations in a file `dg-output-bars`, in the same order in which the `DG_OUTPUTF` macro was called during the recording pass. Then,

`tape-evaluation` $\langle path \rangle$ (6.5)

iterates through the tape file back-to-front and performs the bar value update (2.17). In case the bar value $\bar{a}_{\text{lhs}}$ of the left-hand side is zero, care is taken not to change the bar values $\bar{a}_1$ and $\bar{a}_2$ of the operands even if the partial derivatives are infinite (and the product of zero and infinity would be NaN). Performing (2.18) is not required as indices were not reused. Only chunks of the tape file are loaded into memory, to reduce the memory footprint compared with loading the full tape file at once. In the end, the bar values of the input variables are written to `dg-input-bars` in the same order in which `DG_INPUTF` was called during the recording pass.

Listing 6.2: Extension of the Valgrind function wrapper in Listing 5.3 for the recording pass, new code is highlighted in green. The wrapper records a tape block with the analytical derivative, and disables the recording of the numerical approximation algorithm in the math library implementation. The code is further extended in Listing 7.1.

```c
#include "valgrind.h"
#include "derivgrind.h"
#include <math.h>
#include <stdbool.h>

__attribute__((optimize("O0")))
double I_WRAP_SONAME_FNNAME_ZU(libmZdsoZa, sin) (double x) {
  OrigFn fn;
  VALGRIND_GET_ORIG_FN(fn);
  bool already_disabled = DG_DISABLE(1,0)!=0;
  double ret;
  CALL_FN_D_D(ret, fn, x);
  double ret_d = ret;
  if(!already_disabled) {
    if(DG_GET_MODE=='d'){ /* forward mode */
      double x_d;
      DG_GET_DOTVALUE(&x, &x_d, 8);
      double ret_d = (cos(x)) * x_d;
      DG_SET_DOTVALUE(&ret, &ret_d, 8);
      DG_DISABLE(0,1);
    } else if(DG_GET_MODE=='b') { /* recording pass */
      unsigned long long x_i, y_i=0;
      DG_GET_INDEX(&x, &x_i);
      double x_pdiff, y_pdiff=0.;
      x_pdiff = (cos(x));
      unsigned long long ret_i;
      DG_DISABLE(0,1);
      DG_NEW_INDEX(&x_i,&y_i,&x_pdiff,&y_pdiff,&ret_i,&ret_d);
      DG_SET_INDEX(&ret,&ret_i);
    } /* will add code for the bit-trick finder mode */
  } else {
    DG_DISABLE(0,1);
  }
  return ret;
}

/* ... wrappers for other math.h functions ... */
```

6.7.2. Forward-Mode Tape Evaluation

To perform a tape-based forward-mode evaluation according to Paragraph 2.4.5.c, the user has to store the dot values of the AD inputs in a file `dg-input-dots`, in the same order in which `DG_INPUTF` was called, and invoke

$$\texttt{tape-evaluation } \langle \textit{path} \rangle \texttt{ --forward}. \tag{6.6}$$

This performs (2.15) for all tape blocks front-to-back, and stores the the dot values of the AD outputs in `dg-output-dots`, in the same order in which `DG_OUTPUTF` was called.

6.7.3. Additional Functionality

The tape evaluator program can also be invoked with the argument `--print` to generate a human-readable tabular representation of the tape, as shown in Listing 8.7 in Chapter 8. Also, there is a `--stats` option to output the numbers of blocks with zero, one and two non-zero indices, respectively.

7. Semi-Automatic Detection of Bit-Tricks

As discussed in the previous Chapters 5 and 6, Derivgrind can insert forward-mode or recording-pass instrumentation for all data transfers and real-arithmetic operations that it recognizes in the VEX superblocks presented by the Valgrind core. When the client program performs real-arithmetic operations via the corresponding floating-point instructions on x86 or x86-64, Valgrind usually translates them into the corresponding VEX operations, which are easy to recognize for Derivgrind. (As we noted in Table 4.3, some data transfers using the 80-bit floating-point format are translated to dirty calls in VEX, which are easy to spot as well.)

However, client programs can perform real arithmetic in other ways than by using the corresponding floating-point instruction. We have listed a few of the possible “bit-trick” mechanisms in Section 3.3. They may already be present in the source code (in the style of Listings 3.19 and 3.20), or be introduced by the compiler for “clean” source code (e. g. as in Listings 3.17 and 3.18). Anticipating the observations made while applying Derivgrind to real-world software in Chapters 8 and 10, we can report that Derivgrind genereally handles compiler-generated bit-tricks well, but numerical software sometimes contains hard-coded bit-tricks that Derivgrind’s forward-mode and recording-pass modes do not recognize.

In this chapter, we present a third, *bit-trick-finding* “mode”, which can heuristically detect a set of bit-tricks, and give useful information to localize them in the code of the client program.

7.1. Shadow Data: Activity and Discreteness Flags

Like the forward mode and recording pass, Derivgrind’s bit-trick-finding instrumentation keeps track of shadow data for each piece of data handled by the primal program.

7.1.a. Shadowing Policy. The bit-trick-finder mode uses two layers of shadow storage locations.

- The *lower* layer is used to store *activity flags*. Whenever a bit in a storage location depends on user-declared inputs, the corresponding bit in the lower shadow layer should be 1, otherwise it should be 0.
- The *upper* layer is used to store *discreteness flags*. Whenever a bit in a storage location results from an operation that Derivgrind assumes does not produce

floating-point data, the corresponding bit in the upper shadow layer should be 1, otherwise it should be 0.

In particular, the default shadow data for uninitialized memory is all-zero: Globals and constants in the client program initially do not depend on user-declared input, and whether they are discrete or floating-point data is not known a priori.

As opposed to the forward-mode and recording-pass policies (Paragraphs 5.1.a and 6.1.b), full compliance with this policy even in corner cases is not strictly necessary – false positives or negatives of a heuristic debugging tool are generally more acceptable than wrong derivatives computed in production.

7.1.b. Summary of the Instrumentation. The bit-trick-finding instrumentation propagates the activity and discreteness flags as follows:

- For data transfers between two storage locations, the instrumentation simply transfers the activity and discreteness flags between the respective shadow storage locations.
- For data processing operations, generally, activity bits of the operands propagate to the result in an "infectious" way: If a single activity bit of a single operand is set, all activity bits of the output will be set, unless we know better for the specific operation at hand. The discreteness bits of the result are all set to 0 if the operation is recognized and handled by the forward-mode and recording-pass instrumentations, and to 1 otherwise.

The bit-trick finder generates a warning message with a backtrace if a floating-point operand of a recognized real-arithmetic operation has at least one active and discrete bit. Users can then take a look at the source code of the client program to find out where the active discrete value was produced, and this hopefully leads them to the location of the bit-trick.

7.1.c. Rationale. We have selected this exact form of a heuristic due to the following considerations:

- Tracking whether data is discrete or belongs to a floating-point representation looks like the obvious thing to do. For pre-initialized memory of the program that stores globals and constants, we do not know which parts of these data are floating-point numbers. Also, floating-point data can be legitimately processed by non-floating-point instructions, e. g. while writing an encrypted or compressed file (in any case, the outcome is discrete).

  The only time we can safely identify data as floating-point data is when it is used in, or results from, a floating-point operation. This suggests that the proper choice is to distinguish between discrete data on the one hand, and floating-point data and "do not know" on the other hand. Hence we have a discreteness flag and not a "floating-pointness" flag.

- Tracking activity information is a means of reducing messages about irrelevant bit-tricks: We expect that users run the bit-trick finder because Derivgrind computes wrong derivatives, so they already know which variables are possible AD inputs. They are probably not interested in bit-tricks applied to passive variables because fixing these bit-tricks would not affect the wrong derivatives.

- In addition to the proposed forward propagation of activity flags, one could also propagate indices (as in the recording pass) and only report bit-tricks that can affect user-declared output variables. Implementing this would be rather complex, especially because it would need more than two layers of shadow storage locations and therefore make it necessary to work around the size limit of Valgrind's guest state (Paragraph 4.4.a).

- Valgrind's Memcheck tool[167] (Paragraph 4.1.3.a) warns about uninitialized values when they are used in certain ways, and its warning messages show backtraces of both the use and the origin of the uninitialized value. One could try to employ a similar mechanism in Derivgrind's bit-trick finder tool to hint the user to the location of the bit-trick, and not only to the location where its result is afterwards used by regular real arithmetic. For now, we however prefer to keep the complexity low.

7.2. Instrumentation of Statements and Expressions

For the bit-trick-finder feature, VEX statements are instrumented with *flags-propagating statements* in a very similar fashion as in the recording pass. VEX expressions in a statement might translate to *flags-propagating expressions* in the flags-propagating statements, which, again, look very similar to the dot-value-propagating and index-handling and tape-recording expressions of Chapters 5 and 6, for those VEX expression types that only transfer data. The handling of VEX operation expressions, outlined in Paragraph 7.2.a, is however very different and will be outlined next.

7.2.a. Operations. Like the recording-pass instrumentation of VEX operations (Paragraph 6.4.c), the bit-trick-finder instrumentation looks at recognized SIMD operations as sequences of scalar operations on each individual component. For each floating-point operand of a recognized scalar real-arithmetic operation, their flags-propagating expressions are passed to a dirty call, which checks for bits that are both active and discrete, and prints a user warning with a backtrace in this case. Activity bits in the operands also affect only the activity bits of the respective component of the result.

For operations not handled in the forward mode or recording pass, a default handler determines the activity bits of the result in an infective way fully implemented in VEX, and the discreteness bits of the result are always 1. The C code implementing the handling of VEX operations for the forward mode, recording pass, and bit-trick finder instrumentations are generated from the same Python script (Paragraph 6.4.d), so it is easy to make sure that the three modes recognize the same set of operations as real arithmetic.

7.2.b. Dirty calls. The bit-trick-finder treats the x87-related dirty calls (see Table 4.3) like a recognized floating-point operation: A warning message is raised if the input has a bit that is considered both active and discrete, the activity bits of the result depend on the activity of the operand in an infectious fashion, and discreteness bits of the result are set to zero. For the other dirty calls, activity bits are not propagated (for simplicity) but the discreteness flags are set to 1.

7.2.c. CCalls. Likewise, activity bits propagate through CCalls in an infectious fashion, and the result has all discreteness bits set to one.

7.3. Identifying Input Variables and Accessing Flags

As for the forward and reverse modes, we define monitor commands and client requests to allow the user to access activity and discreteness flags.

7.3.a. Monitor Commands. The monitor command `flagsget` ⟨*address*⟩ ⟨*size*⟩ allows the user to read activity and discreteness flags interactively from a connected debugger session.

7.3.b. Client Requests. The client request macros

```
DG_GET_FLAGS(⟨addr⟩,⟨Aaddr⟩,⟨Daddr⟩,⟨size⟩)
DG_SET_FLAGS(⟨addr⟩,⟨Aaddr⟩,⟨Daddr⟩,⟨size⟩)
```

move activity and discreteness bits describing ⟨*size*⟩-many bytes at ⟨*addr*⟩ between the shadow memory and the memory at ⟨*Aaddr*⟩ and ⟨*Daddr*⟩. We also define a helper macro `DG_MARK_FLOAT(`⟨*var*⟩`)` that sets all activity bits of a floating-point variable *var* to 1, and all discreteness bits to 0. Users can apply this macro to declare AD inputs, i. e. instead of the forward-mode `DG_SET_DOTVALUE` and the recording-pass `DG_INPUTF`.

7.4. Math Function Wrappers

Listing 7.1 shows the final version of Derivgrind's wrapper for the `sin` function in `libm.so*`, built on top of Listing 6.2. In the bit-trick-finding mode, the wrapper treats math functions like elementary real-arithmetic operations: A warning message is issued if the floating-point operand has active and discrete bits; the result is not discrete; and the result is all-active if any bit of the operand is active, and not-active otherwise.

Table 7.1: Regression test results for the bit-trick finder instrumentation. Failures are explained in Section 7.5.

Bit-trick mechanism	Recognized?
`frexp` via incomplete masking (Paragraph 3.3.4.a)	yes
Integer addition to exponent (Paragraph 3.3.4.b)	yes
Rounding via floating-point inaccuracies (Section 3.3.5)	no
Encryption followed by decryption (Section 3.3.6), using the Twofish cipher in CBC mode in `gcrypt`	yes
Compression followed by inflation (Section 3.3.6), using `zlib`	(no)
Multi-mapped memory (Paragraph 4.5.1.e)	no

As the math wrappers for the forward mode and recording pass (Sections 5.5 and 6.6) solve all correctness problems related to unrecognized bit-tricks in the math library implementation, the bit-trick-finding instrumentation suppresses warnings when the thread-local counter `dg_disable[threadID]` is non-zero, i. e., in the call to the original math function between the two macro calls `DG_DISABLE(1,0)` and `DG_DISABLE(0,1)`.

7.5. Evaluation

We have implemented a few "regression tests" to systematically check the bit-trick-finder's response for certain bit-tricks listed in Section 3.3. An additional testcase has been adapted from Listing 3.15, to see if writing floating-point data to a virtual address `*x` and reading it from a different virtual address `*y` of the same portion of physical memory is treated as a binary identity. The results of our tests are shown in Table 7.1. For a practical use case, see Paragraph 10.3.e.

7.5.a. Why is the "rounding" bit-trick not detected? When, according to Section 3.3.5, a large constant T is added to a floating-point number y and then immediately subtracted, this appears like an ordinary sequence of real-arithmetic operations. The bit-trick-finder instrumentation does not realize that these operations come with a floating-point error that has intended real-arithmetic effects.

In fact, computer code that calculates $y \mapsto (y+T)-T$ can be interpreted as an almost everywhere differentiable real-arithmetic function in two possible ways, as an identity $y \mapsto y$ and as a rounding operation – this contradicts the most basic assumption of AD that there is a unique way to look at a computer program as a mathematical function, as stated in Paragraphs 1.1.a and 2.2.c. Derivgrind's derivative computation and bit-trick finding assume the code to represent $y \mapsto y$, while it is actually intended as a rounding operation.

7.5.b. What about the "compression followed by inflation" bit-trick? Whether or not the bit-trick finding mode shows a warning message depends on the version and con-

Listing 7.1: Complete Valgrind function wrapper for the function `sin` from `libm.so*`, extending Listing 6.2 with the bit-trick-finder handling highlighted in green.

```c
#include "valgrind.h"
#include "derivgrind.h"
#include <math.h>
#include <stdbool.h>

__attribute__((optimize("O0")))
double I_WRAP_SONAME_FNNAME_ZU(libmZdsoZa, sin) (double x) {
  OrigFn fn;
  VALGRIND_GET_ORIG_FN(fn);
  bool already_disabled = DG_DISABLE(1,0)!=0;
  double ret;
  CALL_FN_D_D(ret, fn, x);
  double ret_d = ret;
  if(!already_disabled) {
    if(DG_GET_MODE=='d'){ /* forward mode */
      double x_d;
      DG_GET_DOTVALUE(&x, &x_d, 8);
      double ret_d = (cos(x)) * x_d;
      DG_SET_DOTVALUE(&ret, &ret_d, 8);
      DG_DISABLE(0,1);
    } else if(DG_GET_MODE=='b') { /* recording mode */
      unsigned long long x_i, y_i=0;
      DG_GET_INDEX(&x, &x_i);
      double x_pdiff, y_pdiff=0.;
      x_pdiff = (cos(x));
      unsigned long long ret_i;
      DG_DISABLE(0,1);
      DG_NEW_INDEX(&x_i,&y_i,&x_pdiff,&y_pdiff,&ret_i,&ret_d);
      DG_SET_INDEX(&ret,&ret_i);
    } else if(DG_GET_MODE=='t') { /* bit-trick-finding mode */
      DG_DISABLE(0,1);
      unsigned long long xA[2]={0,0}, xD[2]={0,0};
      DG_GET_FLAGS(&x, xA, xD, 8);
      dg_trick_warn_clientcode(xA, xD, 8);
      if(xA[0]!=0||xA[1]!=0) xA[0] = xA[1] = 0xfffffffffffffffful;
      xD[0] = xD[1] = 0;
      DG_SET_FLAGS(&ret, xA, xD, 8);
    }
  } else {
    DG_DISABLE(0,1);
  }
  return ret;
}

/* ... wrappers for other math.h functions ... */
```

figuration of `zlib` (checked with a Ubuntu installation vs. a version built from source). If warning messages are shown, they appear in the `deflate` call and not in the real-arithmetic operation performed on its output. This suggests that they are "false positives", triggered e. g. by a bitwise logical operation that happens to look like a sign bit manipulation (Section 3.3.2) but is actually not real-arithmetic operation, and that the actual bit-trick is not recognized. Our hypothesis for the latter statement is that `zlib`'s compression algorithm, which is called DEFLATE[50] and uses LZ77[201] and Huffman coding[86], mainly rearranges and duplicates bytes in the data stream. E. g., a repeated sequence of floating-point numbers in the original data stream would be stored only once in the compressed data stream and then duplicated during inflation. This affects derivative information but only uses data transfer operations, which are considered "safe" by the bit-trick finding mode.

7.5.c. Why is the "multi-mapped memory" trick not detected? We keep track of shadow data for all virtual addresses used by the primal program; access to shadow data is made using virtual addresses (Paragraph 4.4.c). Different virtual addresses `x` and `y` have different shadow data attached to them, even if they point to the same portion of physical memory as in Listing 3.15. Therefore, writing into `*x` does not affect the activity or discreteness bits of `*y`.

7.6. Alternative: Comparing Dot Values and Difference Quotients

Before we developed the bit-trick-finder instrumentation described in the previous sections, we experimented with an extension of the forward-mode instrumentation (Chapter 5). In addition to propagating dot values through all of the recognized real-arithmetic operations, the tool would record the values and dot values of the results of (most of) these operations. The instrumented client program is run twice, using two slightly different values for the single AD input. These two values should be close enough so that the control flow and the set of performed real-arithmetic operations is exactly the same.

Then, for each real-arithmetic operation, we can compute the difference quotient (2.12) of its results for the two input values, to approximate the derivatives of the intermediate results with respect to the AD input. The two input values must neither be too close nor too far apart for this to work (Paragraph 2.1.d). Then, the approximated derivative can be compared against the two dot values.

Ideally, the dot values and the difference quotients start to differ, i. e. the relative difference starts to significantly increase, precisely after the first unrecognized bit-trick has been performed. In practice, we have successfully used this approach to locate a bit-trick in the particle physics toolkit Geant4[10,12,13], as described in Paragraph 10.3.e. According to our (limited) experience, it required some trial-and-error to find a suitable perturbation of the AD input, and even then there were false positives. In contrast, the instrumentation based on activity and discreteness flags, as described previously, detects the bit-trick without much manual intervention.

8. Validation

Having discussed the ideas and implementation details behind Derivgrind in the previous Chapters 5 to 7, this is the first of three chapters in which we evaluate the novel AD tool. Section 8.1 describes our extensive collection of regression tests, which we use to check the correctness of the computed derivatives for a large number of small test programs. Besides, we apply Derivgrind to two complex programs that evaluate simple arithmetic expressions: The standard Python interpreter CPython in Section 8.2, and the spreadsheet software LibreOffice Calc in Section 8.3. Chapter 9 continues with a performance study based on a numerical solver for Burgers' partial differential equation, and Chapter 10 gives an overview on our applications of AD and Derivgrind to complex simulations used in medical and high-energy physics.

8.1. Regression Tests

We have developed Derivgrind simultaneously with an extensive suite of regression tests. The test system is a Python script that creates and runs many small client programs, considering most of the possible combinations of

- the following arithmetic calculations to be differentiated: elementary operations, calls to math functions, control structures, loops suitable for auto-vectorization, and some OpenMP constructs;
- implemented in the following languages: C, C++, Fortran 90, Python;
- using the following floating-point types: `binary32` (as C/C++ `float`, Fortran `real`, Python `numpy.float32`), `binary64` (as C/C++ `double`, Fortran `double precision`, Python `numpy.float64` and `float`), and the 80-bit x87 type (as C/C++ `long double`)
- compiling C and C++ codes with GCC or Clang compilers, and Fortran codes with GCC; Python scripts are interpreted with the system's `python3` executable, as described in Section 8.2.
- on either the x86 or x86-64 architecure (C, C++, Fortran), or the single one of these architectures for which `python3` was compiled (Python);
- to be run under either the forward or the recording mode of Derivgrind.

Listing 8.1: Sample regression test, checking the correctness of Derivgrind's forward-mode handling of a multiplication of `float`s in C; reformatted.

```
#include <stdio.h>
#include <valgrind/derivgrind.h>

int main(){
  int ret=0;
  float a = 1.0;
  float b = 2.0;
  {
    float _derivative_of_a = 3.0;
    DG_SET_DOTVALUE(&a,&_derivative_of_a,4);
    float _derivative_of_b = 4.0;
    DG_SET_DOTVALUE(&b,&_derivative_of_b,4);
  }
  float c = a*b;
  {
    if(c < 1.9999 || c > 2.0001) {
      printf("VALUES DISAGREE: c stored=%.9f computed=%.9f\n",
        (float)2.0,c);
      ret = 1;
    }
    float _derivative_of_c = 0.;
    DG_GET_DOTVALUE(&c,&_derivative_of_c,4);
    if(_derivative_of_c < 9.9999 || _derivative_of_c > 10.0001) {
      printf("DOT VALUES DISAGREE: c stored=%.9f computed=%.9f\n",
        (float)10.0, _derivative_of_c);
      ret = 1;
    }
  }
  return ret;
}
```

8.1.a. Forward-Mode Regression Tests. Listing 8.1 gives an example of such a client program; the code has been reformatted for the purpose of presentation in this thesis. This particular client program is used to verify the correctness of Derivgrind's forward-mode derivative for a multiplication of two `float`s in C. To this end, the client program

- declares input variables `a` and `b` of type `float`, and initializes them with values specified in the test suite,
- uses client requests to seed their dot values with dot values specified in the test suite,
- performs the arithmetic operation in question,
- checks the value of the result against the correct value stored in the test suite, and prints an error message if both values differ beyond a reasonable tolerance parameter,
- uses a client request to retrieve the dot value of the result and checks it against the correct dot value stored in the test suite, again printing an error message if both values differ too much, and finally
- exits, with an exit status reflecting whether there were error messages.

All forward-mode regression tests stick to this structure regardless of the programming language; see Listing 8.2 for a Fortran, and Listing 8.8 in Section 8.2 for a Python example. For C, C++ and Fortran tests, the test system creates an optimized build of the client program with the usual compiler commands (`gcc`, `g++`, `clang`, `clang++`, `gfortran`), and runs it under Derivgrind. For Python tests, which we explain in more detail in Section 8.2, the test system runs `python3` under Derivgrind, letting it execute a test script like Listing 8.8. In either case, the test system uses the exit status to determine whether the test has passed or failed.

8.1.b. Recording-Mode Regression Tests. Listing 8.3 is the recording-mode variant of Listing 8.1. It uses client requests to declare variables as AD inputs and outputs, and only checks the value of the result. The test system runs the client program under Derivgrind, and then either

- creates a file with the output bar values stored in the test suite, executes the tape evaluator in reverse mode (Section 6.7.1), and checks the input bar value file against the expected derivatives stored in the test suite, or
- creates a file with the input dot values stored in the test suite, executes the tape evaluator in forward mode (Section 6.7.2), and checks the output dot value file against the expected derivatives stored in the test suite.

Again, this procedure also applies to Fortran programs and Python scripts.

Listing 8.2: Fortran 90 version of the regression test in Listing 8.1, reformatted.

```
program main
use derivgrind_clientrequests
use, intrinsic :: iso_c_binding
implicit none
integer :: ret = 0
real, target :: a = 1.0d0
real, target :: b = 2.0d0
block
  real, target :: derivative_of_a = 3.0d0
  call dg_set_dotvalue(c_loc(a), c_loc(derivative_of_a), 4)
end block
block
  real, target :: derivative_of_b = 4.0d0
  call dg_set_dotvalue(c_loc(b), c_loc(derivative_of_b), 4)
end block
block
  real, target :: c; c= a*b
  if(c < 1.9999d0 .or. c > 2.0001d0) then
    print *, "VALUES DISAGREE: c stored=", 2.0d0, " computed=", c
    ret = 1
  end if
  block
    real, target :: derivative_of_c = 0
    call dg_get_dotvalue(c_loc(c), c_loc(derivative_of_c), 4)
    if(derivative_of_c < 9.9999d0 .or. derivative_of_c > 10.0001d0) ↩
        then
      print *, "DOT VALUES DISAGREE: c stored=", 10.0d0, " computed=", ↩
          derivative_of_c
      ret = 1
    end if
  end block
end block
call exit(ret)
end program
```

Listing 8.3: Recording-mode variant of the regression test in Listing 8.1, reformatted.

```
#include <stdio.h>
#include <valgrind/derivgrind.h>

int main(){
  int ret=0;
  float a = 1.0;
  float b = 2.0;
  {
    DG_INPUTF(a);
    DG_INPUTF(b);
  }
  float c = a*b;
  {
     if(c < 1.9999 || c > 2.0001) {
       printf("VALUES DISAGREE: c stored=%.9f computed=%.9f\n",(float)↩
           2.0,c); ret = 1;
     }
     DG_OUTPUTF(c);
  }
  return ret;
}
```

8.1.c. Results. Derivgrind passes almost all of these tests. Rare failures can be unanimously attributed to the limitations explained in this thesis:

- Valgrind not supporting AVX on x86 (Paragraph 4.5.1.d);
- Derivgrind not supporting multi-threading (Section 5.2.2) in general, and bit-tricks in an OpenMP implementation (Listing 3.22) in particular;
- bit-tricks in NumPy math functions on `binary32` arguments in CPython on x86-64 (see Appendix A.2); and
- bit-tricks used by GCC to implement `rint` (Listing 3.21).

8.2. Application to CPython

8.2.a. Can Derivgrind differentiate Python scripts? Some of the regression tests in Section 8.1 involve Python scripts. Derivgrind is of course not directly applied to Python code; the actual client program is the Python interpreter running these scripts:

$$\texttt{valgrind --tool=derivgrind python3}\ \langle\textit{Python script}\rangle\ \langle\textit{arguments for Python script}\rangle \qquad (8.1)$$

Under the hood, a reasonable Python interpreter should represent the values of Python `float` variables as `binary64`s, and perform Python arithmetic operations through the corresponding instructions. It is these operations of the Python interpreter that Derivgrind instruments.

8.2.b. CPython. On many Linux systems in general and in our test container in particular, the default Python interpreter `python3` is provided by the "Python reference implementation" CPython. CPython is typically installed on the system by a software package manager as a pre-compiled binary package. Thus, the source code of CPython is generally not stored on the system. To understand how we are still able to declare AD inputs and outputs, we have to elaborate on the *Python C modules* mechanism first.

8.2.c. Python C Modules. Python C modules are shared objects compliant with the Python/C API. They can be built from a C/C++ source file with an ordinary build toolchain plus the Python headers. The C/C++ source must define specific functions required by the Python/C API. Wrapper generators like SWIG and pybind11[99] make this much easier.

When the Python interpreter reads a Python statement `import` ⟨*module*⟩, it searches several directories, including those specified by the environment variable `PYTHONPATH`, for pure Python modules ⟨*module*⟩`.py` and ⟨*module*⟩`/__init__.py`, but also for shared objects with the name ⟨*module*⟩`.so` (or variants of it). If such a shared object is found, the Python interpreter loads it at run-time, uses the Python/C API to query for the symbols that the module would like to expose to the Python side, and the respective Python names. Python calls to functions of the Python/C module are then dispatched to the appropriate functions in the shared object.

As an example, core components of the NumPy package[78] for scientific computing in Python have been implemented in C and wrapped via the Python/C API. NumPy implements the scalar Python types `numpy.float32` and `numpy.float64` as C `float`s and `double`s, and dispatches many math functions to the C math library as well. If the Python script involves NumPy types and arithmetic, it's the compiled C code of NumPy that Derivgrind instruments.

We use the Python C/API to make C implementations of client requests callable from Python. To this end, we have developed a Python C module `derivgrind` as a side component of the Derivgrind package. Table 8.1 lists the Python functions defined by the module. They dispatch to C code that contains the corresponding client request macros.

Note that the functions `set_dotvalue`, `inputf` and `mark_float` do not modify the AD meta-data of their arguments, as their C equivalents do. Python `float`s are immutable, so functions cannot change the value of a `float` argument. Even though we would only need to write to the shadow memory, we cannot be sure if our Python module could get a pointer to the original `binary64` storage behind the Python `float`, and not just a copy, next to other problems. Therefore, the Python functions modify the AD meta-data of a local copy and return a copy of it. Thus, they are used like this:

```
x = derivgrind.set_dotvalue(x, 1.0).
```

8.2.d. Forward-Mode Examples. In the Python script in Listing 8.4, `x` and `y` are ordinary Python `float`s. Even if the CPython instance running the script is executed under

Table 8.1: Python functions provided by the Python C module `derivgrind`.

Forward mode:	
`set_dotvalue(x,d)`	Calls `DG_SET_DOTVALUE` to seed a copy of `x` with the dot value `d`, and returns a copy of this.
`get_dotvalue(x)`	Calls `DG_GET_DOTVALUE` on a copy of `x` and returns the dot value.
Recording pass:	
`inputf(x)`	Calls `DG_INPUTF` on a copy of `x`, and returns a copy of this.
`outputf(x)`	Calls `DG_OUTPUTF` on a copy of `x`.
Bit-trick finder:	
`mark_float(x)`	Calls `DG_MARK_FLOAT` on a copy of `x`, and returns a copy of this.
`get_flags(x)`	Calls `DG_GET_FLAGS` on a copy of `x` and returns a list containing the values of the activity and discreteness flags.

Listing 8.4: When this Python script is run by CPython interpreter executing under Derivgrind in the forward mode according to (8.1), it prints the correct derivative $e^{1.0} \cdot 10.0$. Client requests to seed and read the dot values are injected into CPython by the Python C module `derivgrind`, whose functions are listed in Table 8.1.

```python
import math
import derivgrind
x = derivgrind.set_dotvalue(1.0,10.0)
y = math.exp(x)
print(derivgrind.get_dotvalue(y))
```

Derivgrind via (8.1), `print(x)` gives the value `1.0` and `type(x)` gives `<class 'float'>`, because Derivgrind's instrumentation of the client program is (almost) transparent and the `DG_SET_DOTVALUE` client request only affects Derivgrind's data structures (setting the dot value to 10.0) which are separate from the memory of the client program. The dot value of `y` is printed as `27.18281828459045`, matching the analytic derivative $\frac{\partial e^x}{\partial x} \cdot \dot{x} = e^{1.0} \cdot 10.0$.

Listing 8.5 is a variant of Listing 8.4, using the exponential function of `numpy` instead of `math`. We observe the same behavior as for Listing 8.4. However, when `numpy.float64` is replaced by `numpy.float32`, the dot value of `y` is printed as `0.0`. This is because on our test system, NumPy, rather than using the C math library, implements several mathematical functions on `binary32` arguments on its own, using bit-tricks that are not supported by Derivgrind. See Appendix A.2 for details. This problem does not appear when NumPy has been built for a target architecture that supports only the most basic SSE instructions, as specified in Paragraph A.2.d.

Listing 8.5: Variant of Listing 8.4, using the exponential function of `numpy` instead of `math`.

```
import numpy
import derivgrind
x = derivgrind.set_dotvalue(1.0,10.0)
y = numpy.exp(numpy.float64(x))
print(derivgrind.get_dotvalue(y))
```

Listing 8.6: Variant of Listing 8.5 with the reverse-mode functions of the Python C module `derivgrind`.

```
import numpy
import derivgrind
x = derivgrind.inputf(1.0)
y = numpy.exp(numpy.float64(x))
derivgrind.outputf(y)
```

8.2.e. Recording-Mode Examples. The Python script in Listing 8.6 uses the recording-mode client requests to declare `x` as an AD input and `y` as an AD output. With Derivgrind's recording-pass instrumentation, CPython records the tape displayed in Listing 8.7. The `binary64` behind the Python variable `x` is assigned the index `1`, the tape block for index `2` is written by the math wrapper, and as always, an extra index is allocated for every declaration of an AD output.

When `numpy.float64` is replaced by `numpy.float32` in Listing 8.6, Derivgrind fails because of an internal Valgrind error related to an upper threshold on the number of temporaries (Paragraph 4.4.b). Starting Valgrind with an additional flag

`--vex-guest-max-insns=10` (8.2)

fixes this issue, letting the process run successfully and record a tape with 27 blocks. However, the index of `y` is zero and thus any tape evaluation returns a zero derivative. As in the previous Paragraph 8.2.d, building NumPy with only the basic SSE instructions fixes the problem.

8.2.f. Regression Tests. Coming back to the integration of Python tests into the regression test suite, the Python script in Listing 8.8 is analogous to the C program in Listing 8.1, using functions of the Python C module `derivgrind.so` instead of the client request macros. Also, for the tests that involve NumPy types, variables are replaced by arrays of 16 elements, so operators and math functions are vectorized.

8.2.g. Summary. With the example of CPython, we have demonstrated that Derivgrind can be applied to large programs that were compiled elsewhere and installed as a binary package, without any source code at hand. Insertions into the source code of CPython

Listing 8.7: Derivgrind tape recorded for CPython running Listing 8.6.

```
|-------------------|-------------------|-------------------|
|                 0 |                 0 |                 0 |
|             dummy |      0.000000e+00 |      0.000000e+00 |
|-------------------|-------------------|-------------------|
|                 1 |                 0 |                 0 |
|             input |      0.000000e+00 |      0.000000e+00 |
|-------------------|-------------------|-------------------|
|                 2 |                 1 |                 0 |
|                   |      2.718282e+00 |      0.000000e+00 |
|-------------------|-------------------|-------------------|
|                 3 |                 2 |                 0 |
|            output |      1.000000e+00 |      0.000000e+00 |
|-------------------|-------------------|-------------------|
```

Listing 8.8: Python version of the regression test in Listing 8.1. As regression tests based on NumPy types use vectorized operations and perform every initialization and check 16 times, the script has been shortened at the marked locations.

```python
import numpy as np
import derivgrind as dg
ret = 0
a = np.empty(16,dtype=np.float32)
b = np.empty(16,dtype=np.float32)
a[0] = 1.0
a[1] = 1.0
...
a[15] = 1.0
b[0] = 2.0
...
a[0] = dg.set_dotvalue(a[0], 3.0)
...
b[0] = dg.set_dotvalue(b[0], 4.0)
...
c = a * b
derivative_of_c = np.empty(16,dtype=np.float32)
if c[0] < 1.9999 or c[0] > 2.0001:
  print("VALUES DISAGREE: c[0] stored=", 2.0, "computed=", c[0])
  ret = 1
...
derivative_of_c[0] = dg.get_dotvalue(c[0])
if derivative_of_c[0] < 9.9999 or derivative_of_c[0] > 10.0001:
  print("DOT VALUES DISAGREE: c[0] stored=", 10.0, "computed=", ↩
      derivative_of_c[0])
  ret = 1
...
exit(ret)
```

Table 8.2: LibreOffice Calc macros defined by Listing B.4.

Forward mode:	
`SetDotValue`	Seeds the numerical value of the current cell with the dot value of the right neighbor cell.
`GetDotValue`	Overwrites the right neighbor cell with the dot value of the current cell.
Recording pass:	
`InputF`	Marks the numerical value of the current cell as an AD input variable.
`OutputF`	Marks the numerical value of the current cell as an AD output variable.

are not necessary because CPython exposes Python variables to user-supplied C code at runtime. In a small number of cases, the computed derivative is incorrect because of unsupported bit-tricks in a library (NumPy) used by the program; identifying and fixing the issue requires access to the source code of the library.

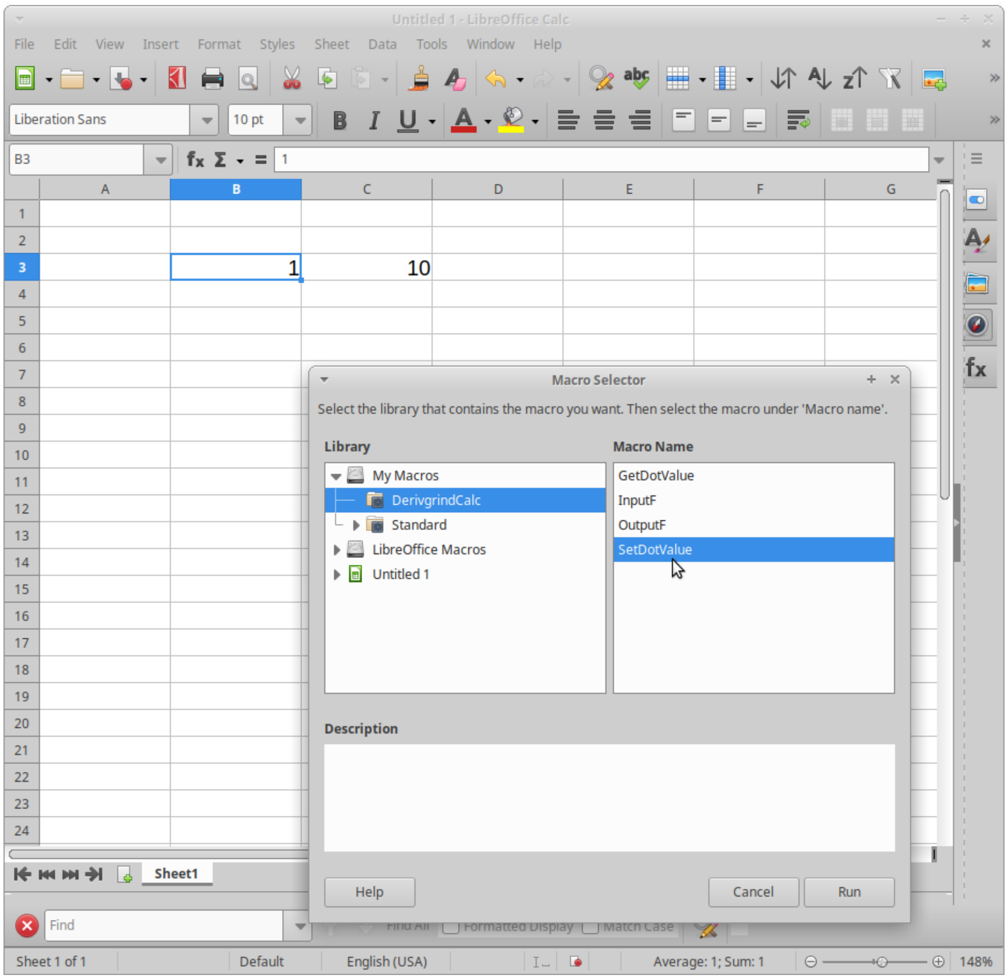
8.3. Application to LibreOffice Calc

In Section 8.2, we have applied Derivgrind to a Python interpreter, which performed floating-point arithmetic as specified by a Python script. This way we validated Derivgrind for a large program, without requiring access to its source code. In this section, we push this idea further, with another "calculator" that can be considered even more complex and also allows for nice visual demonstrations: We apply Derivgrind to the spreadsheet program *LibreOffice Calc*[56,115], which performs floating-point arithmetic as specified by formulas entered into a graphical user interface (GUI).

LibreOffice Calc is a component of the free and open-source office suite LibreOffice, and was installed in version 7.3.7.2 on a Ubuntu 22.04 system via the `apt-get` package manager along with a few LibreOffice development and debugging symbol packages.

8.3.a. LibreOffice Calc Macros. Similar to the Python C module mechanism of CPython, LibreOffice Calc provides the user with capabilities to load and execute user-supplied code. To this end, LibreOffice Calc *macros* can be written in either the *StarOffice Basic* language, or in Python. Python macros seem to be executed by the same process that performs Calc's floating-point arithmetic, and are thus a suitable location to make client request, using the Python C module `derivgrind.so` presented in the previous Section 8.2.

8.3.b. Procedure. The user has to copy the Python file in Listing B.4 into one of the macro search directories of LibreOffice Calc. The file defines the four Python Calc macros listed in Table 8.2; see Appendix B.3 for details of the implementation. The Python code



Select the library that contains the macro you want. Then select the macro under 'Macro name'.

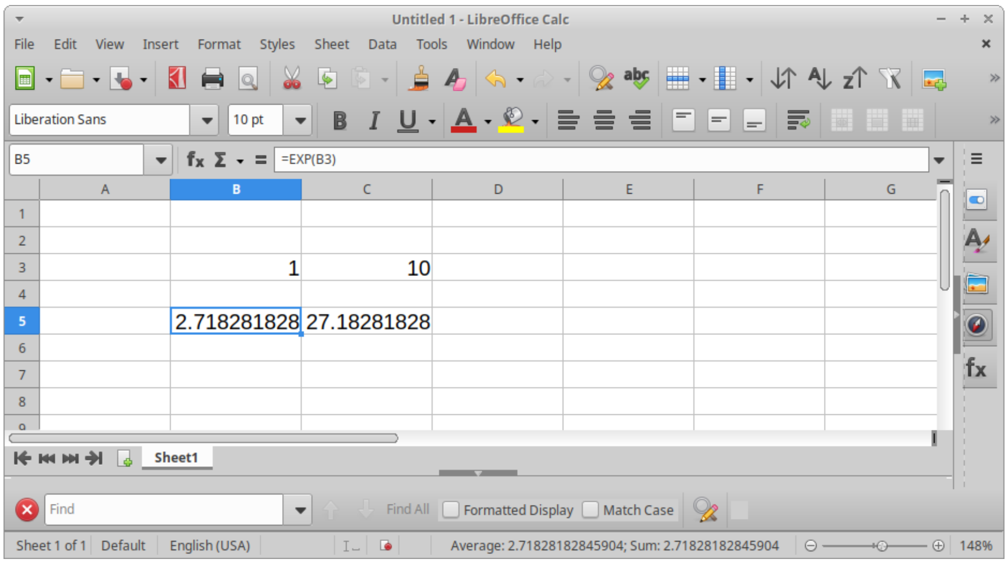
Library	Macro Name
My Macros	GetDotValue
DerivgrindCalc	InputF
Standard	OutputF
LibreOffice Macros	SetDotValue
Untitled 1	

Figure 8.1: LibreOffice Calc macro selector dialog.

imports the Python C module `derivgrind`, so a directory containing `derivgrind.so` must be included in the environment variable `PYTHONPATH`. Similar to (8.1), the user runs LibreOffice Calc under Derivgrind via

```
valgrind --tool=derivgrind [-record=⟨path⟩]
                  /usr/lib/libreoffice/program/soffice.bin --calc    (8.3)
```

in either the forward or recording mode. Here, `soffice.bin` is the executable that is finally started as a child process. It is possible to put the `libreoffice` script instead, if an additional argument `--trace-children=yes` is supplied to Valgrind. As soon as the LibreOffice Calc GUI is ready, the user can type in values and formulas, and run the appropriate two of the four client request macros via the *Run Macro...* dialog shown in Figure 8.1.


Untitled 1 - LibreOffice Calc
File Edit View Insert Format Styles Sheet Data Tools Window Help
Liberation Sans
10 pt
Sheet1
Find
Find All
Formatted Display
Match Case

B5 =EXP(B3)

	A	B	C	D	E	F	G
1							
2							
3		1	10				
4							
5		2.718281828	27.18281828				
6							
7							
8							

Sheet 1 of 1 | Default | English (USA) | Average: 2.71828182845904; Sum: 2.71828182845904 | 148%

Figure 8.2: Forward-mode derivative of LibreOffice Calc.

8.3.c. Results. We have validated the forward- and reverse-mode derivatives of LibreOffice Calc evaluating some simple expressions like `EXP(B3)`, `SIN(B3)` and `SQRT(TAN(B3))`, where the cell `B3` contains the value 1.0 and the dot value 10.0 (in forward mode), or has been declared as an AD input (in recording mode).

As an example for the forward mode and the expression `EXP(B3)`, Figure 8.2 shows the computed dot value $\frac{\partial\,\texttt{EXP(B3)}}{\partial\,\texttt{B3}} \cdot \dot{\texttt{B3}}$.

Listing 8.9 shows extracts of the tape recorded by Derivgrind while LibreOffice Calc evaluated `EXP(B3)`. The dependency of the output variable with respect to the input variable is given by a chain of blocks with the same partial derivatives as in the tape recorded for CPython (Listing 8.7). Derivgrind recorded many more blocks for LibreOffice Calc, which indicates that Calc performs many additional arithmetic operations whose operands depend on the input variable, but whose results have no effect on the output variable. The first of these blocks stem from a function `doubleToString` in LibreOffice's System Abstraction Layer (SAL) for platform-independent strings, which contains, e. g., calls to the C math function `log10` and floating-point divisions.

Listing 8.9: Extracts of the Derivgrind tape recorded for LibreOffice Calc evaluating `EXP(B3)`, where the content of cell `B3` has been declared as an AD input.

```
|-------------------|-------------------|-------------------|
|                 0 |                 0 |                 0 |
|             dummy |      0.000000e+00 |      0.000000e+00 |
|-------------------|-------------------|-------------------|
|                 1 |                 0 |                 0 |
|             input |      0.000000e+00 |      0.000000e+00 |
|-------------------|-------------------|-------------------|
|                 2 |                 1 |                 0 |
|                   |      2.718282e+00 |      0.000000e+00 |
|-------------------|-------------------|-------------------|
|                 3 |                 2 |                 0 |
|                   |      1.597680e-01 |      0.000000e+00 |
|-------------------|-------------------|-------------------|
...
|-------------------|-------------------|-------------------|
|               365 |                 2 |                 0 |
|            output |      1.000000e+00 |      0.000000e+00 |
|-------------------|-------------------|-------------------|
...
|-------------------|-------------------|-------------------|
|               5a4 |               5a3 |                 0 |
|                   |      1.000000e+01 |      9.590451e-01 |
|-------------------|-------------------|-------------------|
```

9. Performance

Putting universal applicability without a lot of manual intervention before performance, Derivgrind prioritizes the design goals for AD tools (Paragraph 1.1.c) in a way that is different from most of the existing source-code-based AD tools. Nevertheless, in this chapter, we quantify and compare run-time and memory performance characteristics of Derivgrind using a numerical benchmark.

After introducing the benchmark in Section 9.1, we separately study the performance in the forward mode (Section 9.2) and with the recording pass (Section 9.3), comparing to native execution without AD and to the operator-overloading AD tool CoDiPack[158].

9.1. Benchmark

9.1.a. Primal Program. We adapt our benchmark primal program from previous studies on the performance of CoDiPack[157–159] and its add-on OpDiLib[32]. The C++ code approximately solves the two-dimensional coupled Burgers' partial differential equation (PDE)[19,26,200]

$$\begin{aligned}
\frac{\partial u}{\partial t}(t,x,y) + u(t,x,y)\cdot\frac{\partial u}{\partial x}(t,x,y) + v(t,x,y)\cdot\frac{\partial u}{\partial y}(t,x,y)& \\
= \frac{1}{R}\cdot\left(\frac{\partial^2 u}{\partial x^2}(t,x,y) + \frac{\partial^2 u}{\partial y^2}(t,x,y)\right)&, \qquad (9.1)\\
\frac{\partial v}{\partial t}(t,x,y) + u(t,x,y)\cdot\frac{\partial v}{\partial x}(t,x,y) + v(t,x,y)\cdot\frac{\partial v}{\partial y}(t,x,y)& \\
= \frac{1}{R}\cdot\left(\frac{\partial^2 v}{\partial x^2}(t,x,y) + \frac{\partial^2 v}{\partial y^2}(t,x,y)\right)&, \qquad (9.2)
\end{aligned}$$

for functions $u, v : [0,T]\times[0,1]^2 \to \mathbb{R}$, subject to initial conditions

$$u(0,x,y) = u_0(x,y) \qquad (9.3)$$

$$v(0,x,y) = v_0(x,y) \qquad (9.4)$$

for all (x,y) in $[0,1]^2$, and boundary conditions

$$u(t,x,y) = \frac{x+y-2xt}{1-2t^2} \qquad (9.5)$$

$$v(t,x,y) = \frac{x-y-2yt}{1-2t^2} \qquad (9.6)$$

for all $t > 0$ and (x, y) on the boundary of $[0,1]^2$. Intuitively, (9.1) and (9.2) describe the evolution of “densities” u and v on a two-dimensional domain $[0,1]^2$ in time. The rate $\frac{\partial w}{\partial t}(t, x, y)$ of change of either density w at a particular point (x, y) at time t is given as a combination of two effects:

- “Convection” of the density $w = u, v$ according to a velocity field $\binom{u}{v}$ contributes a change rate of
$$-\begin{pmatrix} u \\ v \end{pmatrix} \cdot \begin{pmatrix} \partial w/\partial x \\ \partial w/\partial y \end{pmatrix}, \tag{9.7}$$
whose negative appears on the left hand side of (9.1) and (9.2).
- “Diffusion”: Intuitively, an uneven density distribution w leads to a diffusive flux proportional to $\binom{\partial w/\partial x}{\partial w/\partial y}$ with a negative proportionality factor $-\frac{1}{R}$. This flux field accumulates density according to the negative divergence,
$$-\frac{\partial}{\partial x}\left(-\frac{1}{R}\frac{\partial w}{\partial x}\right) - \frac{\partial}{\partial y}\left(-\frac{1}{R}\frac{\partial w}{\partial y}\right)$$
which shows up as a source term on the right hand side of (9.1), (9.2).

In the numerical solver, u and v are discretized in space and time. I. e., values $u_{i,j}^{(k)}$, $v_{i,j}^{(k)}$ of these functions on a two-dimensional rectangular grid $\{\binom{i\cdot\Delta x}{j\cdot\Delta x} : i, j = 0, \dots, \frac{1}{\Delta x}\}$ are stored to approximate u, v for particular points in time $k\cdot\Delta t$, $k = 0, \dots, \frac{T}{\Delta t}$. To translate the partial differential equations (9.1) and (9.2) to the discretized versions of $w = u, v$, derivatives are approximated by difference quotients:

$$\begin{aligned}
&\frac{w_{i,j}^{(k+1)} - w_{i,j}^{(k)}}{\Delta t} \\
&\quad + u_{i,j}^{(k)} \cdot \begin{Bmatrix} \frac{w_{i,j}^{(k)} - w_{i-1,j}^{(k)}}{\Delta x}, & \text{if } u_{i,j}^{(k)} \geq 0 \\ \frac{w_{i+1,j}^{(k)} - w_{i,j}^{(k)}}{\Delta x}, & \text{otherwise} \end{Bmatrix} + v_{i,j}^{(k)} \cdot \begin{Bmatrix} \frac{w_{i,j}^{(k)} - w_{i,j-1}^{(k)}}{\Delta x}, & \text{if } v_{i,j}^{(k)} \geq 0 \\ \frac{w_{i,j+1}^{(k)} - w_{i,j}^{(k)}}{\Delta x}, & \text{otherwise} \end{Bmatrix} \\
&\qquad = \frac{1}{R} \cdot \frac{w_{i-1,j}^{(k)} + w_{i+1,j}^{(k)} + w_{i,j-1}^{(k)} + w_{i,j+1}^{(k)} - 4w_{i,j}^{(k)}}{\Delta x^2}.
\end{aligned} \tag{9.8}$$

Note that for the spatial derivatives in the convection terms, either a forward or backward difference quotient is used, depending on the sign of the convective flow. Intuitively, such an *upwind scheme* ensures that “upstream” and not “downstream” data is taken into account to compute the future evolution of u and v at each point;[113] this is important and careless choices can easily lead to unphysical solutions. The diffusion term is discretized with second-order central difference quotients.

Values at $k = 0$ are determined from the initial conditions (9.3), (9.4), and are then successively obtained for larger k by solving the linear equation (9.8) for $w_{i,j}^{(k+1)}$. Thus, the numerical solver only uses the four basic arithmetic operations addition, subtraction, division and multiplication, with the possibility for autovectorization. In the end, we compute the sum of the Euclidean norms of the vectors $(u_{ij}^{(k)})$ and $(v_{ij}^{(k)})$ after the final timestep $k = \frac{T}{\Delta t}$. This way, also a square root appears in the function to be differentiated.

Listing 9.1: Implementation of the update (9.8) in the Burgers' benchmark. `u` and `v` store all the entries $u_{i,j}^{(k)}$ and $v_{i,j}^{(k)}$ in a flat one-dimensional vector; the values of u and v at (i, j), $(i + 1, j)$, $(i - 1, j)$, $(i, j + 1)$, $(i, j - 1)$ are accessible in `u` and `v` at `index`, `index_xp`, `index_xm`, `index_yp`, `index_ym`. The function is called two times per time step, with `w_t` being either `u` or `v`; the function computes the respective field in the next time step into `w_tp`. The constant `props.dTbyDX` is $\frac{\Delta t}{\Delta x}$ and `props.dTbyDX2` is $\frac{\Delta t}{\Delta x^2}$.
The code has been provided by Max Sagebaum.

```
inline void updateField(Number *w_tp, const Number *w_t, const Number *↩
    u, const Number *v, const Settings& props) {
  // w_t + u*w_x + v*w_y = 1/R(w_xx + w_yy);
  Number velX;
  Number velY;
  Number vis;
  for (size_t j = props.innerStart; j < props.innerEnd; ++j) {
    for (size_t i = props.innerStart; i < props.innerEnd; ++i) {
      size_t index = i + j * props.gridSize;
      size_t index_xp = index + 1;
      size_t index_xm = index - 1;
      size_t index_yp = index + props.gridSize;
      size_t index_ym = index - props.gridSize;

      if (u[index] >= 0.0) {
        velX = u[index] * (w_t[index] - w_t[index_xm]);
      } else {
        velX = u[index] * ( w_t[index_xp] - w_t[index]);
      }
      if (v[index] >= 0.0) {
        velY = v[index] * (w_t[index] - w_t[index_ym]);
      } else {
        velY = v[index] * (w_t[index_yp] - w_t[index]);
      }

      vis = w_t[index_xp] + w_t[index_xm] + w_t[index_yp] + w_t[↩
          index_ym] - 4.0 * w_t[index];
      w_tp[index] = w_t[index] - props.dTbyDX * (velX + velY) + props.↩
          oneOverR * props.dTbyDX2 * vis;
    }
  }
}
```

9.1.b. System Setup. Generally, we consider $2^3 = 8$ setups, using the GCC 10.2.1 (`g++`) and Clang 11.0.1 (`clang++`) compilers, for the x86-64 (no flag) and x86 (`-m32`) ISA, with full (`-O3`) and without (`-O0`) compiler optimizations. The client program was compiled and executed on an exclusive 64-bit Intel Xeon Gold 6126 processor at 2.6 GHz in the Elwetritsch cluster at the University of Kaiserslautern-Landau, with a sufficient amount of main memory. For the recording-pass measurements, the tape file was placed in the ramdisk directory `/dev/shm`.

9.1.c. Measurement Details. Our time measurements refer to the difference in the system time retrieved by the client program right before and after solving the PDE, if not noted otherwise. This way, we exclude the constant-time startup and finalization of Derivgrind and the shadow memory tool from the time measurement. Averages were taken over 100 (`-O3`) or 10 (`-O0`) measurements in the forward mode, and 25 (`-O3`) or 5 (`-O0`) measurements in the reverse mode.

Our memory measurements refer to the maximum resident set size (RSS) reported by the GNU `time` command, if not noted otherwise.

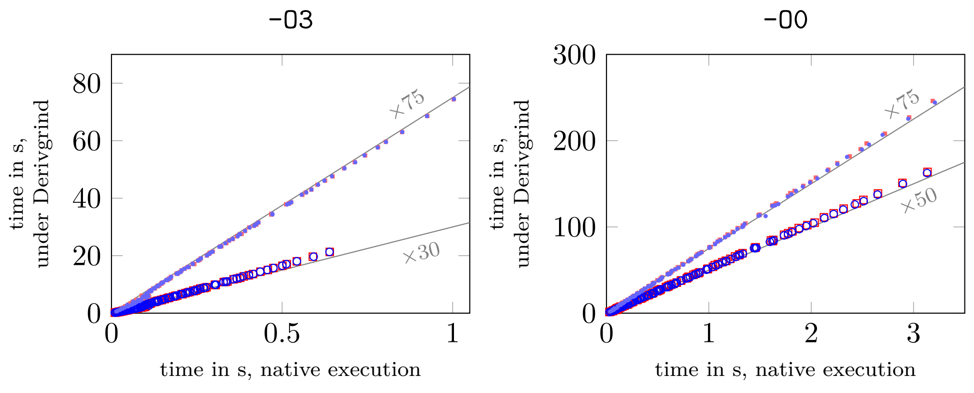
9.2. Forward Mode

In the forward mode, we use AD to differentiate the output with respect to a value added to all components of the initial state, i. e. the sum of the derivatives with respect to all $u_{i,j}^{(0)}$ and $v_{i,j}^{(0)}$.

9.2.a. Run-Time Performance. Figure 9.1 displays the effect of Derivgrind's forward mode on the run-time of the PDE solver. Each marker in the plots represents a problem instance with an $n_x \times n_x$ grid and n_t time steps, for $n_x = 100, 120, \dots, 500$ and $n_t = 100, 200, \dots, 500$. The plots show that Derivgrind slows down the PDE solver by a factor that is essentially independent from n_x and n_t, and varies between 30 and 75. The best factor of about 30 is reached for the practically most relevant case of an optimized build on x86-64. As Derivgrind's instrumentations of the various VEX constructs differ in complexity, and probably offer a different amount of opportunities for optimizations by the Valgrind core, it is natural that the slow-down factor depends on the "mixture" of instructions produced by the compiler.

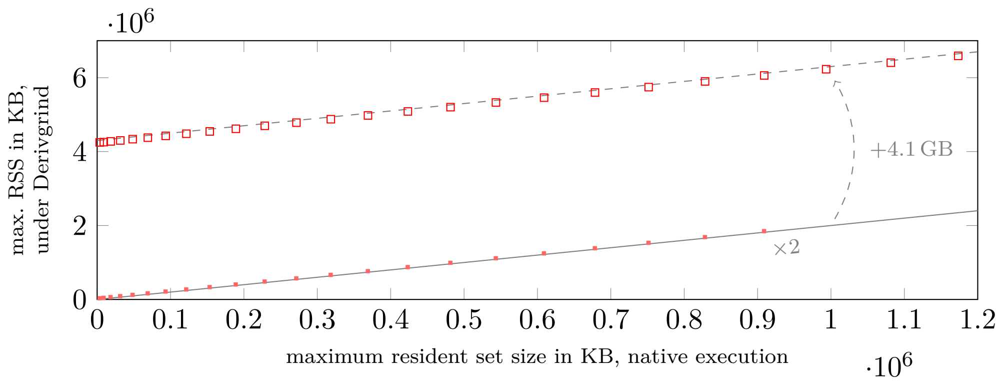
For comparison, when running the benchmark with CoDiPack's forward mode, the largest slow-down factor measured by us on the setups with `-O3` is approximately 3.3.

9.2.b. Main Memory Performance. Figure 9.2 displays the effect of Derivgrind on the required memory. We consider problem instances on an $n_x \times n_x$ grid and $n_t = 4$ time steps, for $n_x = 200, 400, \dots, 5000$, as the memory consumption hardly depends on n_t. As the plot shows, Derivgrind doubles the memory consumption, in addition to a constant reservation of about 4.1 GB on x86-64 and 20 MB on x86 for the default configuration of the shadow memory tool. On x86-64, the shadow memory tool needs much more



□ GCC on x86-64. ▪ GCC on x86. ○ Clang on x86-64. • Clang on x86.

Figure 9.1: Derivgrind's effect on the run-time of the Burgers benchmark.



□ GCC on x86-64. ▪ GCC on x86.

Figure 9.2: Derivgrind's effect on the maximum resident set size of the Burgers benchmark.

memory for its internal data structures (Paragraph 4.4.d); changing their layout, the constant allocation can be brought below 0.1 GB with a minor run-time penalty.

Compiling the program with Clang instead of GCC, and/or disabling optimizations (`-O0`), has no significant effect on the required memory.

9.3. Reverse Mode

In the reverse mode, we declare all components $u_{i,j}^{(0)}$ and $v_{i,j}^{(0)}$ of the initial state as AD inputs.

9.3.a. Run-Time of the Tape Recording Figure 9.3 displays the effect of Derivgrind's recording-pass instrumentation on the run-time, each marker representing a problem instance with $n_x = 100, 120, \ldots, 400$ and $n_t = 100, 200, 300, 400$.

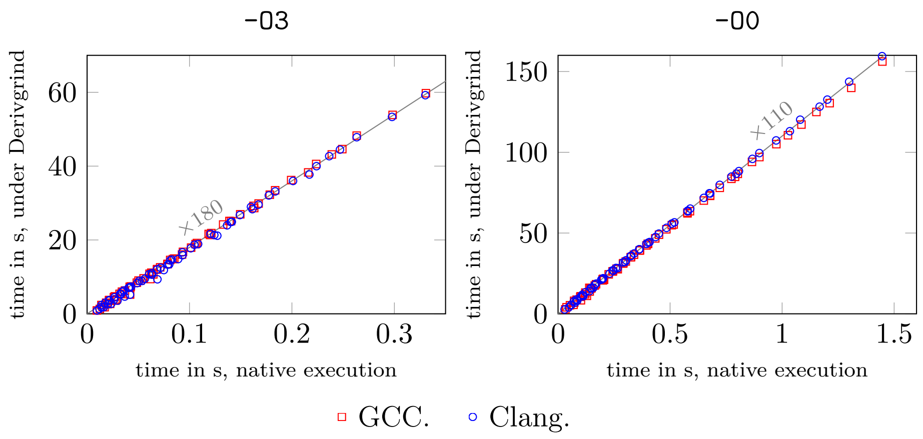


Figure 9.3: Effect of Derivgrind's recording-pass instrumentation on the run-time of the Burgers benchmark.

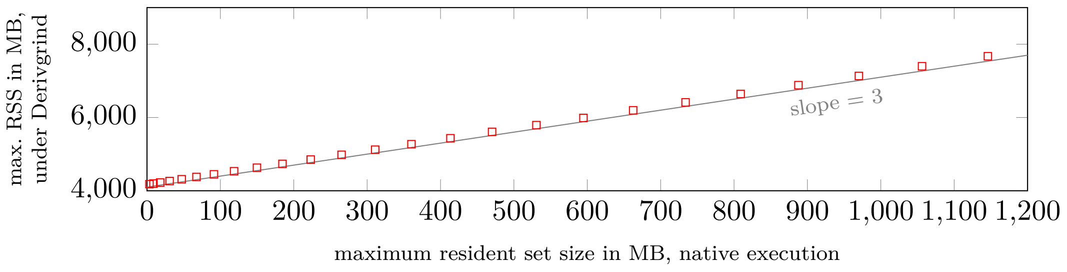


Figure 9.4: Effect of Derivgrind's recording-pass instrumentation on the maximum resident set size of the Burgers benchmark, in a setup using GCC and `-O3`. RAM or disk space occupied by the tape file is not included here.

Asymptotically, Derivgrind scales the run-time of the client program by a proportionality factor of about 180 for optimized and 110 for unoptimized builds.

9.3.b. Memory Complexity of the Tape Recording, Excluding Tape Figure 9.4 displays Derivgrind's scaling of the memory consumption in terms of maximum RSS. The RSS adds up allocations made by the client code, the Valgrind core, and the Derivgrind tool including the shadow memory tool and the tape buffer, but excludes any RAM space required to store the tape file on the ramdisk. We considered problem instances with $n_x = 200, 400, \dots, 5000$ and $n_t = 4$. Only the results for GCC with `-O3` are shown, as changing to Clang and/or `-O0` had only minor effects.

The slope of 3 is consistent with the fact that two bytes of shadow memory are allocated for every byte of memory that the client uses. The instance-independent constant allocation of about 4.1 GB on x86-64 is mainly occupied by the shadow memory tool. Configuring it differently, we can decrease the constant allocation to below 0.1 GB, at the price of slightly increasing the run-time.

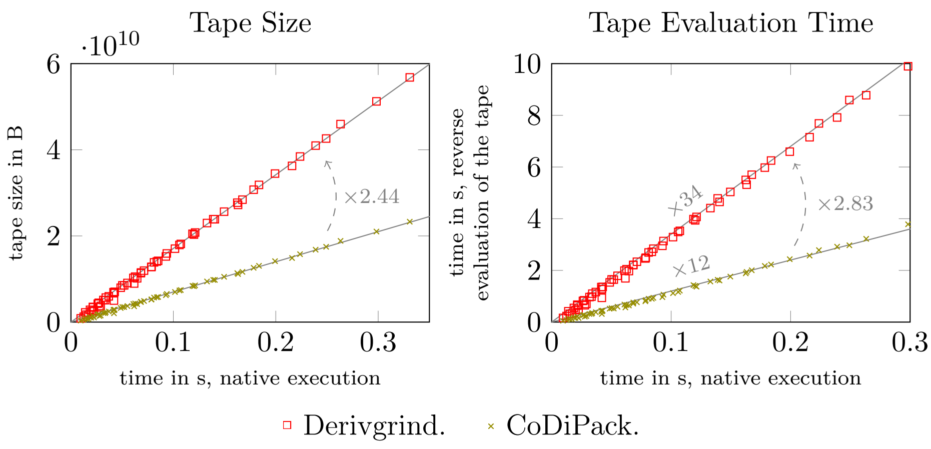


Figure 9.5: Tape sizes and reverse tape evaluation run-times for Derivgrind and CoDiPack, plotted against the native run-time of the client program built with GCC and `-O3`.

9.3.c. Tape Size The left part of Figure 9.5 shows that the ratio between the sizes of the tapes recorded by Derivgrind and CoDiPack, for one run of the Burgers' PDE benchmark compiled with GCC and `-O3`, is around 2.44 across all problem instances. Likewise, the numbers of pairs of non-zero indices and partial derivatives on the respective tapes consistently follow a ratio close to 1.66. The dominant part of the PDE solver's real arithmetic are four C++ statements in the body of a nested loop, which Derivgrind and CoDiPack represent with 448 B vs. 184 B of tape space ($\frac{448\,\mathrm{B}}{184\,\mathrm{B}} = 2.43\ldots$) and 25 vs. 15 pairs of non-zero indices and partial derivatives ($\frac{25}{15} = 1.66\ldots$), respectively. CoDiPack needs less tape space and less pairs because it can make use of expression templates, and allows for a more flexible tape layout. The ratios could be even larger if the right-hand sides in the C++ code were more complex. On the other hand, this result also tells us that Derivgrind does not record a significant amount of unnecessary operations on x86-64. These observations are also made with the Clang compiler and/or `-O0`.

9.3.d. Run-Time of the Tape Evaluation As the reverse tape evaluation procedure is simple and unrelated to the machine code of the client program, we can expect its run-time performance to catch up with the state of the art, in which the run-time is memory bandwidth bound.[32,177] The right side of Figure 9.5 compares the reverse tape evaluation run-times of Derivgrind and CoDiPack for the same set of problem instances considered in the recording-pass run-time measurements. We observe that Derivgrind's tape evaluation procedure, taking about 2.8 times longer than CoDiPack's, is not much worse given that the tape is 2.4 times longer.

The Burgers' benchmark case has a high proportion of floating-point operations involving active variables. In codes that perform more non-floating-point or passive operations, we expect a smaller tape size to be recorded per second of native run-time. Then, the ratio between the reverse tape evaluation run-time and the native run-time drops below the value of 34 found in Figure 9.5. When the full AD workflow involves a single tape recording but many tape evaluations, as discussed in Paragraph 2.4.5.c, this lower factor dominates the full AD run-time.

9.3.e. Performance Study on x86 On x86, we observed much higher run-time scaling factors up to about 1500 and similar memory scaling factors of about 3. The preceding statements about tape length and correctness apply to small problem instances on x86 as well. We could not test large instances because CoDiPack stores the tape in the memory of the client process, which is limited to about 3 GB on x86.

10. Derivatives in Medical and Particle Physics

Our motivation to build a machine-code-based AD tool has its roots in fruitful discussions with medical and particle physicists. AD has proven effective for the training of machine learning models[22] and for numerical optimization tasks in computational fluid dynamics[11] and various other application domains – yet, a lot of effort is still needed to make AD well-known and accessible as a standard tool for engineering design processes across many industries, such as, for instance, medical and particle physics. Along with colleagues in the MODE collaboration[23,51] sharing the vision to enable gradient-based optimization in the design of particle physics detectors and for physics analyses, we developed the idea of differentiating a complex particle transport code like Geant4[10,12,13], which has over one million lines of C++ code. As our overviews on different source-code-based AD tools has shown in Sections 2.5.2, 2.6 and 2.7, source-code-based AD tools are typically not "automatic" for the entire C++ language standard. Therefore, applying a source-code-based AD tool to complex scientific code, developed for years without AD in mind, can turn out time-consuming; exploratory studies tend to be uneconomical. Feeling the need for an AD tool that requires as little integration efforts as possible, we started to think about ways to perform AD on the machine code level and eventually developed Derivgrind.

In this chapter, we apply Derivgrind to the *GATE software for numerical simulations in medical imaging and radiotherapy*[100] and to a simple calorimeter simulation similar to Geant4's *TestEm3* example. Both applications are based on the *Geant4 toolkit for the simulation of the passage of particles through matter.*[10,12,13]

We give some context on the medical imaging setup in Section 10.1, and analyze three sub-steps for numeric differentiability in Section 10.2 – while this is not strictly necessary to understand the applications of Derivgrind, we would like to give a broader overview on differentiability aspects in proton computed tomography and on the overall scientific environment in which this thesis was written, and present a few less polished results of our work.

In Section 10.3, we describe the application of Derivgrind to GATE. The simplified medical imaging setup features a considerably complex geometry, but we only simulate a single incoming particle with the goal to validate derivatives of some hit coordinates with respect to the energy of the incoming particle. This way, we demonstrate that Derivgrind can be used to solve the technical challenge of applying AD to Geant4.

In practice, Geant4 simulations are conducted for a large number of particles, and the actual output is given, e. g., by statistics of the respective Geant4 outputs. This introduces mathematical challenges that need to be tackled before the derivatives can be used for design optimization. Our work in that regard is summarized in Section 10.4, closing with an application of Derivgrind to a setup tightly related to Geant4's TestEm3 calorimetry example.

10.1. Introduction to Proton Computed Tomography

10.1.1. Computed Tomography

10.1.1.a. X-ray CT. Computed tomography (CT) is a medical and industrial imaging modality that provides a three-dimensional image of the distribution of X-ray absorption rate (radiodensity) in the scanned object. In medical diagnosis, CT images convey information because the radiodensity, expressed in terms of Hounsfield units, varies across different tissue types and materials. Conventional CT scanners send many X-ray "beams" through the object, at various positions and in various directions, and measure the respective attenuations. For every beam, its attenuation is the cumulative effect of the radiodensities along the beam, leading to a linear relation between the (logarithmic) attenuation and the radiodensities of the voxels of the CT image. With many beams and thus many linearly independent equations, it is possible to reconstruct the unknown radiodensities from the measured attenuations. Since its inception in the 1970s,[83] CT has quickly become widely available to patients, with around 250 CT examinations per 1000 people in the US every year.[143]

10.1.1.b. Proton CT. List-mode proton CT (pCT) follows a similar idea, but relies on individual energetic protons instead of X-ray beams, which have different ways of interaction with matter. Regarding the matter of the detector, this means that the particle detectors used in proton CT scanner prototypes use different technologies; we describe one possible prototype in Section 10.1.3. Regarding the matter of the scanned object, pCT measures a different property than the radiodensity, namely, the *relative stopping power* (RSP). In addition to losing kinetic energy according to the RSP, protons are scattered significantly, so besides the proton positions and energy losses, their directions must be measured behind the scanned object. Furthermore, the number of individual protons required for pCT is usually way higher than the number of distinct beams used in X-ray CT. Though pCT was already proposed by Cormack[43] in 1963, developments are still in a prototyping stage.[49,55,87,102,124,126,132,148,155,161,162] We are engaged in pCT research through the Bergen pCT collaboration[14].

Interest in this novel imaging technology is based on the fact that the RSP distribution in a body region, which can only be estimated from X-ray CT Hounsfield units but is measured directly by pCT, is required for *proton radiotherapy* treatment planning. After a digression on this innovative form of cancer treatment in Section 10.1.2, we give more details on the hardware and software setup in Section 10.1.3.

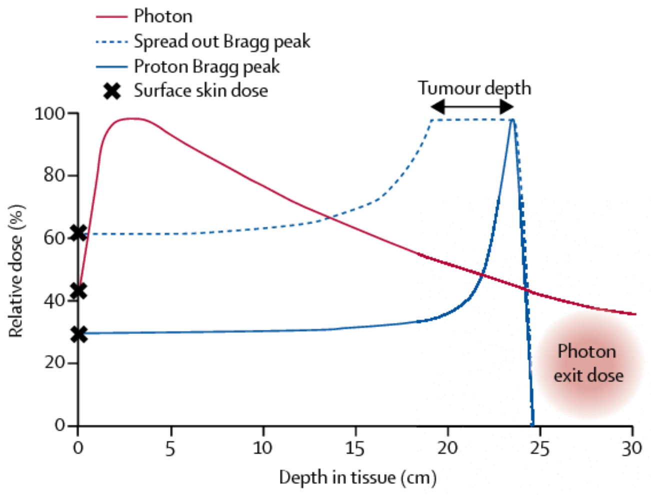


Figure 10.1: Depth-dose curve for X-ray photon and proton beams, reproduced from Leeman et al.[112] Unlike photons, proton beams yield a concentrated dose deposition at a certain depth in the tissue. Choosing a range of initial energies, the Bragg peaks can be spread out to cover the entire tumor region.

10.1.2. Proton Therapy

10.1.2.a. Cancer. Cancer is a group of diseases with the defining feature that abnormal cells grow uncontrollably beyond their usual boundaries, invading adjacent tissue and possibly spreading to other parts of the body; eventually, they account for one in three premature deaths world-wide in 2018.[193] The major treatment options attempt to eliminate cancer cells from the body, by means of

- medication "poisoning" cancer cells in chemotherapy, or enhancing the patient's immune rejection of tumor-associated antigens[82] in immunotherapy,
- surgery, i. e. mechanical removal of tissue containing cancer cells, and
- ionizing radiation that damages the DNA and thus causes cell death.[107]

10.1.2.b. Proton Therapy. Out of these options, in the following we focus on radiation therapy with external beams of energetic charged particles like protons or heavy ions. Figure 10.1, reproduced from Leeman et al.[112], qualitatively shows the dose distributed by a proton beam with a fixed initial energy suited for radiotherapy, depending on the distance that the proton has travelled in water. Most of the dose is concentrated around the *bragg peak* in some particular depth. X-ray beams, in contrast, deposit their energy in a more continuous way shortly after entering the body. The energy loss of protons in other materials, such as body tissue, looks similar to Figure 10.1. The depth of the Bragg peak is related to the beam energy and the RSP of the material along the path.

More and more hadron treatment facilities are constructed on a world-wide scale. It is primarily the existence of the Bragg peak what makes hadron therapy such a promising

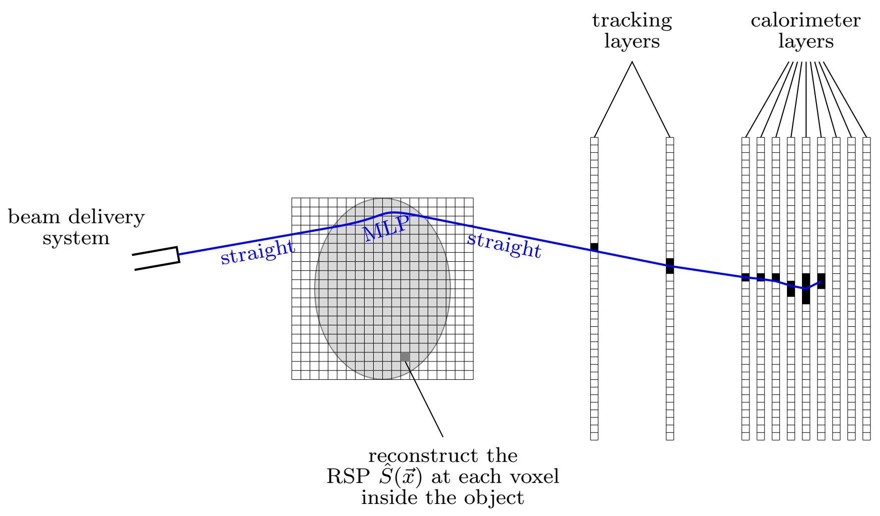


Figure 10.2: Schematic figure of the scanning process with the DTC of the Bergen pCT collaboration, cited from Aehle et al.[3,6]

alternative to conventional X-ray radiotherapy: The dose concentration offers a chance to reduce the dose deposited in the surrounding healthy tissue.[43,190] Many clinical studies have demonstrated the effectiveness of proton therapy for the treatment of various cancers.[16,20,116,199]

10.1.2.c. Where does Proton CT come in? As a consequence of the dose concentration in the Bragg peak, it is of critical importance to select the right beam energy such that the Bragg peak is located in the correct depth. Treatment planning thus relies on an accurate three-dimensional RSP image of the patient. In the state of the art, the RSP is approximated based on single- or dual-energy X-ray CT images; as Hounsfield units and RSP are related to different properties of matter, this indirect approach comes with an uncertainty of up to 3 %[144,192,197] while a direct measurement has been shown to be more accurate.[48,196]

10.1.3. Hardware and Software Setups for Proton CT

10.1.3.a. Detector Hardware. The pCT scanning system designed by the Bergen pCT collaboration[14] is based on a high-granularity digital tracking calorimeter (DTC) sensor that consists of two *tracking* and 41 *calorimeter* layers of 108 ALPIDE (ALICE pixel detector) chips[9] each. After traversing the patient, energetic protons will activate pixel clusters around their tracks in each layer until they are stopped, as shown in Figure 10.2. While many prototypes reported in the literature use an additional pair of front trackers,[55,87,126,132,161,162] the DTC setup infers the positions and directions of entering protons from the beam delivery monitoring system.

10.1.3.b. Detector Simulation. In the development phase and for design optimization studies, the collaboration uses a computer simulation of the DTC instead of the real hardware. At its core, the GATE software[100] based on the Geant4 toolkit[10,12,13] is used for a Monte-Carlo (MC) simulation of the interactions of individual protons with the phantom and the detector. GATE outputs the exact coordinates and energy depositions of hits of the protons in the layers of the DTC. In a post-processing step, this data is converted to the information which ALPIDE pixels have been activated during each detector read-out cycle.

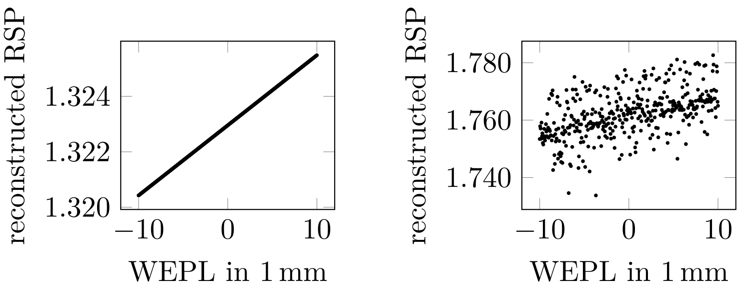
10.1.3.c. Track Reconstruction. Such layer-wise binary activation images from hundreds of protons per read-out cycle, either produced by the real hardware or a simulation, are used to reconstruct the protons' paths and ranges through the detector, and thus their residual direction and energy after leaving the patient. First, neighbouring activated pixels are grouped into *clusters* per layer and read-out cycle. The proton's coordinate is given by the cluster's center of mass and its energy deposition is related to the size of the cluster.[150,174] In the *tracking* step, a track-following procedure[173] attempts to match clusters in neighbouring layers if they likely belong to the same particle trajectory, as indicated by a minimal angular deflection in each layer.[152,153] The energy of the proton can be estimated from the energy depositions per layer by a fit of the Bragg-Kleeman equation of Bortfeld;[34,151] the energy loss is usually expressed in terms of the *water-equivalent pathlength* (WEPL) that would cause the same energy loss on average.

10.1.3.d. Tomographic Reconstruction. Based on this data from various beam positions and directions, a *model-based iterative reconstruction* (MBIR) tomography algorithm like ART, DROP[149] or least-squares conjugate gradient (LSCG) can be used to reconstruct the three-dimensional RSP image of the patient. Mathematically speaking, these algorithms basically perform the task of approximatively solving a (typically overdetermined) linear system of equations $Ax = b$ where

- $x \in \mathbb{R}^{n_\text{voxels}}$ is the vector of RSP values to be determined,
- $b \in \mathbb{R}^{n_\text{protons}}$ is the vector of measured or simulated proton energy losses in terms of WEPL, and
- $A \in \mathbb{R}^{n_\text{protons} \times n_\text{voxels}}$ is the matrix whose entry $A_{i,j}$ represents how much the energy loss of proton track i is affected by the RSP in voxel j (i. e. something like an intersection length).

10.2. Differentiability in a Proton CT Software Pipeline

A long-term research goal of MODE is to optimize particle detectors; in the context of pCT, this means to select a detector geometry and beam parameters, as well as algorithmic parameters in the tracking and reconstruction procedures, in a way to maximize an objective function based on image quality, accuracy, and deposited dose.



(a) DROP, linear. (b) LSCG, noisy.

Figure 10.3: Dependency of the reconstructed RSP at a particular voxel on the WEPL of a particular track.

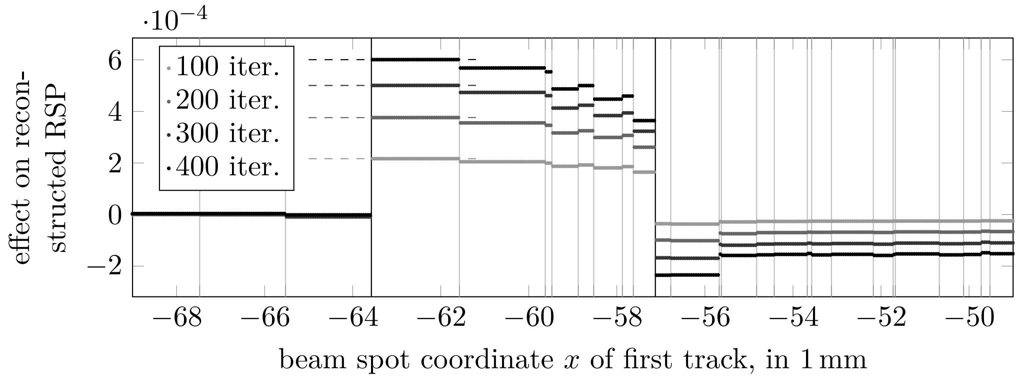
In Aehle et al.[3,6], as a preparation to possible future applications of AD to substeps of the pCT pipeline, we have numerically analyzed for three of them whether they are differentiable, by observing how single output variables react when single input variables are perturbed. Our results are summarized in the following.

10.2.a. Differentiability of the Tomographic Reconstruction. As we stated above in Paragraph 10.1.3.d, mathematically speaking, MBIR algorithms solve a linear system $Ax = b$. We now look separately at derivatives of x with respect to b and A.

Differentiating components of x as AD outputs with respect to components of b as AD inputs can be done analytically: E. g. in the forward mode, $\dot{x}$ can be determined from $\dot{b}$ by solving $A\dot{x} = \dot{b}$. Exploiting structures like this is advisable for performance reasons, to reduce the tape size in the reverse mode, and also to improve numerical accuracy, as outlined in Paragraph 2.2.d. The performance aspect is particularly relevant in this example, as the reconstruction normally runs on a GPU, where AD is technically more difficult to implement, and unoptimized black-box reverse-mode AD hardly exploits the GPU performance.[129] Figure 10.3 illustrates the accuracy aspect for our proton CT reconstruction code: When one entry of b is changed slightly, an entry of x found by the LSCG solver can change a lot.

Differentiating x with respect to components of A is a different story, as the linearity argument does not apply. One of our experiments showed that when a proton track position is perturbed, the reconstructed RSP is a piecewise differentiable function, but the gradients do not reflect the large-scale behaviour very well for a standard configuration of the reconstruction algorithm. As Figure 10.4 shows, this has to do with rounding operations used in the code to determine which voxels were traversed by which proton path in a yes-or-no-fashion, and gets better when a smoother, distance-based "fuzzy voxels" approach is used to compute A from the proton positions and directions.

10.2.b. Differentiability of the Track Reconstruction. The clustering and tracking subprocedure operates a lot on discrete data like pixel activation images, assignments to clusters and matchings between clusters. Effectively, it is part of a sequence of steps that



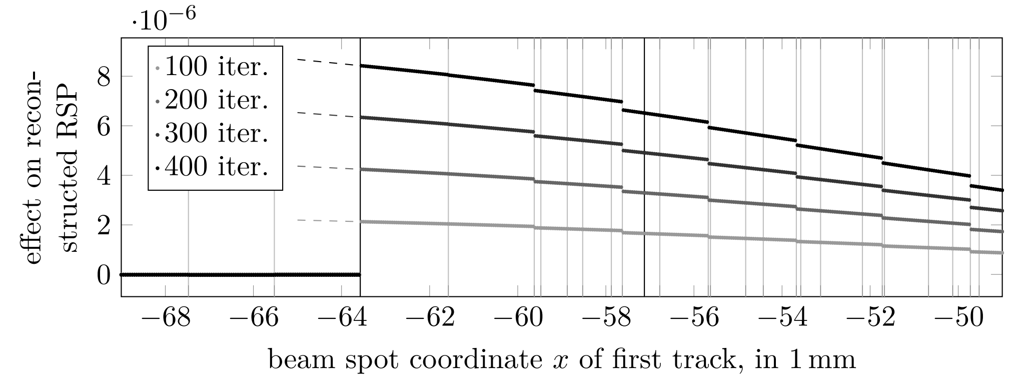


Figure 10.4: Dependency of a reconstructed RSP value on a coordinate of a single proton track, for multiple numbers of DROP iterations and 250 000 simulated proton histories; see Aehle et al.[3,6] for details. Vertical lines indicate where the set of traversed voxels changes. Entries of the system matrix A were determined by a discrete incidence-based approach in the top plot, and a smoother distance-based approach in the bottom plot.

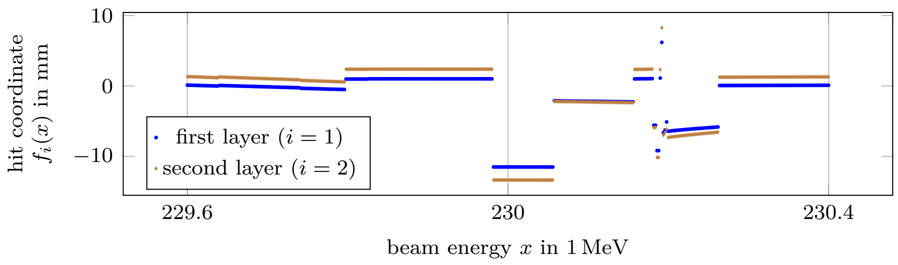


Figure 10.5: Graph of the function that we apply AD to, from Aehle et al.[7]

first convert continuous Geant4 output to discrete detector data, and then convert those back to continous data of proton paths. Further research might replace this sequence of steps by a differentiable model that simply passes the floating-point information through, possibly adding some errors.

10.2.c. Differentiability of the Detector Simulation. For the MC simulation with GATE/Geant4, we have analyzed how the energy depositions and certain position coordinates of a single proton in the first and second tracking layer depend on the beam energy $x \approx 230\,\text{MeV}$. The seed of the pseudo random number generator (RNG) used by the MC simulation has been fixed (correlated sampling[121]).

It has previously been observed, with a more complex setup, that GATE, in version 9.1, produced different simulation outputs across multiple runs even when the seed was fixed. This kind of non-determinism is clearly a bug, and needed to be addressed in order to be able to use the word “function” and attempt to differentiate it. With Valgrind’s memory checking tool Memcheck (see Paragraph 4.1.3.a), we discovered several locations in the code where the control flow depended on the value of uninitialized variables. After our fixes had been applied in version 9.2, GATE behaved deterministically. Figure 10.5 shows plots of a position coordinate computed by GATE version 9.2, indicating that the dependency has jumps but is differentiable in between.

As these jumps look unphysical, we further analyzed two of them, with a methodology and observations described in the following. With GDB, we have set a breakpoint in the RNG code and supplied GDB commands so that whenever the breakpoint was encountered, GDB printed a backtrace. After masking numbers and pointers, we obtained a record of the control flow at locations where random numbers are relevant. We created such a record for four different values of the input x, two on either side of the discontinuity, which were so close to the discontinuity that the records matched for values of x on the same side. For values of x on different sides of the discontinuity, the records started to differ at some point because, as we determined next, a comparison between two almost equal floating-point numbers had different outcomes.

Such a local deviation of the control flow can lead to a different number of RNG calls being made. As a consequence, the subsequent program execution sees a shifted, and thus entirely different, sequence of random numbers. This has nearly the same effect as choosing a different random seed from the beginning, and obviously leads to a different output value. Using the words of Baeten et al.[18], the MC simulation *decorrelates* easily.

This suggest that the discontinuities are, at least in part, an artifact of the way how the RNG is used. One idea to reduce the number of jumps is therefore to re-seed the RNG at well-defined locations, using a random sequence of seeds generated beforehand; however, we have not tested this. In fact, numerical experiments on a simpler detector setup suggest that the jumps are not a problem for AD (Section 10.4).

Between the jumps, the dependency of the position coordinates on the beam energy seems to be differentiable. As the Geant4 toolkit is widely used in particle and medical physics, making derivatives of Geant4 available could have a significant impact for these communities. However, both technical and mathematical challenges are to be expected when differentiating Geant4. In the next Section 10.3, we apply Derivgrind to Geant4 in order to demonstrate that the technical challenge is not a fundamental blocker. We further comment on our work regarding the mathematical challenges in Section 10.4.

10.3. Applying Derivgrind to GATE and Geant4

10.3.a. Technical Challenges. Geant4 is a huge codebase with over one million lines of C++ code. As source-code-based AD tools are typically not fully automatic for the entire language standard, Geant4's size and complexity can turn seemingly little restrictions into a massive amount of work required before being able to obtain algorithmic derivatives. Furthermore, the resulting changes are unlikely to be quickly merged back into the mainline version of Geant4, leading to time-consuming maintenance. This is different for machine-code-based AD tools like Derivgrind, as we show in this section (adapted from Aehle et al.[7]) using the setup described next.

10.3.b. Setup. We consider a setup related to the pCT scanning process in Section 10.1.3, provided by the Bergen pCT collaboration: A single proton from a beam source with beam energy x passes through a digital model of a human head and enters the DTC, where it hits a number of tracking layers until it is stopped. The first component of the coordinate vector of the hits in the first two tracking layers shall be called $f_1(x)$ and $f_2(x)$. The graphs of $f_1(x)$, $f_2(x)$ for x running from 229.6 MeV to 300.4 MeV, plotted in Figure 10.5, look piecewise differentiable.

10.3.c. Numerical Derivatives. Markers in Figure 10.6 show the central difference quotient

$$\tfrac{f_i(x_0+h)-f_i(x_0-h)}{2h} \tag{10.1}$$

around $x_0 = 230\,\mathrm{MeV}$ for various values of h. Mathematically, this difference quotient converges to the derivative $\frac{\partial f_i}{\partial x}(x_0)$ in the limit $h \to 0$ (as already stated in Paragraph 2.1.d). On a computer however, floating-point inaccuracies become large for small h; note that the output values are only stored in `binary32` precision by GATE. For h larger than about 0.02 MeV, the difference quotient (10.1) is entirely off, because $f_{\mathrm{pos},1}$ and $f_{\mathrm{pos},2}$ have a jump near 229.98 MeV.

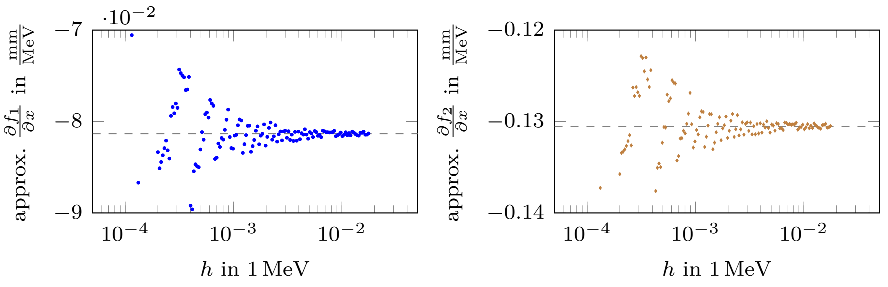


Figure 10.6: Central difference quotients $\frac{f_i(x_0+h)-f_i(x_0-h)}{2h}$ (blue and brown markers) around $x_0 = 230\,\text{MeV}$, and the derivative computed with Derivgrind (dashed line) after changing `G4Log` to `log`. Cited from Aehle et al.[7]

10.3.d. Insertion of Client-Request Macros. Listings 10.1 and 10.2 show our insertions of forward- and recording-mode client requests, respectively, into the source code of GATE. We have slightly refactored the source code beforehand for the purpose of presentation. In the first code block of either listing, GATE reads the energy x from the configuration file and sets the respective property of the beam source object; the inserted code seeds this AD input (forward mode) or declares it as an AD input (recording mode). The second code block is run whenever a particle hits a layer, to assemble output data of GATE. We have inserted code that extracts the first coordinate of the hit position, and reads the dot value of this AD output variable (forward mode) or declares it as an AD output variable (recording mode).

Additionally, the header `derivgrind.h` must be included in the two modified source files, and only they need to be recompiled.

10.3.e. Bit-Trick in Geant4. GATE's intermediate calculations involve Geant4, and Geant4 defines and uses an alternative math function `G4Log` to numerically approximate the natural logarithm $\log z$ for $z \in \mathbb{R}$, using an approximation algorithm adapted from the VDT math library[154]. The implementation of this algorithm uses a bit-trick which Derivgrind does not support; more details can be found in Appendix A.1.

We were first hinted to the existence of a bit-trick in Geant4 by the fact that forward-mode algorithmic derivatives were not matching numerical derivatives. We then localized and identified the bit-trick in the code by comparing dot values with numerical derivatives after each recognized real-arithmetic operation (Section 7.6). The bit-trick-finding heuristic that we finally implemented in Chapter 7, based on discreteness and activity bits, points at `G4Log` right with its first warning message.

We eliminated the bit-trick by replacing `G4Log` by a call to the standard C `log` function.

10.3.f. Results. Running GATE under Derivgrind reproduces the original output of GATE, interleaved with additional output from Derivgrind. In the forward mode, the output also contains the sought derivatives. In the reverse mode, the derivatives are obtained with the tape evaluator from the tape file and input/output index files, as

Listing 10.1: Insertions into the source code of GATE for forward-mode differentiation. They seed the input variable (beam energy), and print the dot value of the output variables (hit positions). Additionally, the header `derivgrind.h` must be included.

```
    if (command == pEnergyCmd) {
      double energy = pEnergyCmd->GetNewDoubleValue(newValue);
+     double one = 1.0;
+     DG_SET_DOTVALUE(&energy,&one,sizeof(double));
      pSourcePencilBeam->SetEnergy(energy);
    }
```

```
    if (m_rootHitFlag) m_treeHit->Fill();
+   float pos = *(float*)(m_treeHit->GetBranch("posX")->GetAddress());
+   float pos_d;
+   DG_GET_DOTVALUE(&pos,&pos_d,sizeof(float));
+   std::cout << "pos_d=" << pos_d << "\n";
```

Listing 10.2: Insertions into the source code of GATE for reverse-mode differentiation. They declare the input variable (beam energy) and output variables (hit positions). Additionally, the header `derivgrind.h` must be included.

```
    if (command == pEnergyCmd) {
      double energy = pEnergyCmd->GetNewDoubleValue(newValue);
+     DG_INPUTF(energy);
      pSourcePencilBeam->SetEnergy(energy);
    }
```

```
    if (m_rootHitFlag) m_treeHit->Fill();
+   float pos = *(float*)(m_treeHit->GetBranch("posX")->GetAddress());
+   DG_OUTPUTF(pos)
```

Table 10.1: Numerical and automatic derivatives of f_1 and f_2 at $x_0 = 230\,\mathrm{MeV}$.

Differentiation method	approximation of $\frac{\partial f_i}{\partial x}$ in $\frac{\mathrm{mm}}{\mathrm{MeV}}$	
	$i = 1$	$i = 2$
Central difference quotient		
… for $h = 0.01\,\mathrm{MeV}$	-0.0812531	-0.130463
… for $h = 0.005\,\mathrm{MeV}$	-0.0811577	-0.130272
… for $h = 0.001\,\mathrm{MeV}$	-0.0815392	-0.131130
Derivgrind, original Geant4		
… forward mode	-0.0685116	-0.113841
… reverse mode	$-1.72 \cdot 10^{8}$	$5.36 \cdot 10^{13}$
Derivgrind, `G4Log` $\rightsquigarrow$ `log`		
… forward mode	-0.0813391	-0.130524
… reverse mode	-0.0813391	-0.130524

usual.

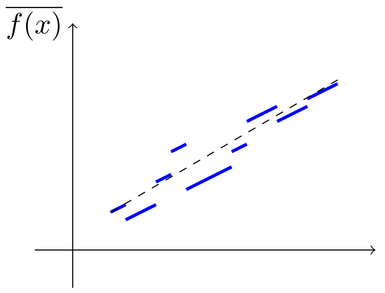
Table 10.1 lists the computed derivatives. Without removing the bit-trick related to `G4Log`, the forward-mode automatic derivatives deviate from the difference quotients by about 15 %, while the reverse-mode derivatives are completely off. With this fix, the automatic derivatives computed by Derivgrind's forward and reverse mode agree, and we indicated them by horizontal lines in Figure 10.6. As these lines are surrounded from both sides by the markers indicating difference quotients, the automatic derivatives are either entirely correct, or at least their deviation from the true derivative is small compared to the inherent variance of difference quotients.

The run-time, measured for a release-mode build with the GNU `time` command on an exclusive 2.6 GHz Intel Xeon Gold 6126 node at the University of Kaiserslautern-Landau, goes up from around 12 s in native execution to 13 min in the forward mode, which is a factor of 65. Derivgrind's recording takes about 24 min, corresponding to a factor of 120, to record a tape of 25 MB whose reverse evaluation takes about 0.05 s.

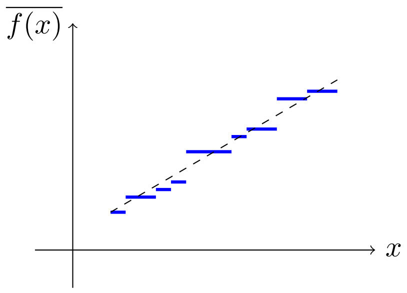
10.4. Outlook: Mathematical Challenges

As shown in Section 10.3, Derivgrind greatly simplifies the *technical challenge* of integrating AD into Geant4. In addition, *mathematical challenges* have to be dealt with before algorithmic derivatives can be used, e. g., for gradient-based design optimization. In this section, we collect our results in that regard; see Aehle et al.[8] for more details.

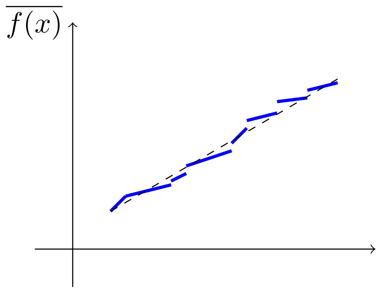
10.4.a. Local and large-scale slopes. The function to be differentiated, as shown in Figure 10.5, is piecewise differentiable: When the AD inputs change, the output of the MC simulation evolves both in a differentiable way and via jumps. The same is true for any objective function computed from the output. In the easiest case, a MC simulation



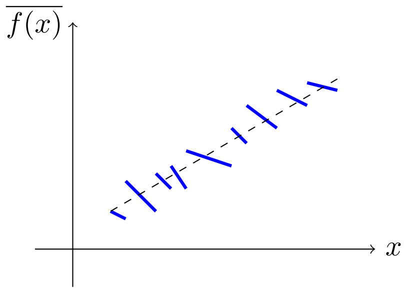
(a) Differentiable evolution dominates, jumps cancelling out on the large scale.



(b) Jumps dominate, differentiable evolution cancelling out.



(c) Mixture of differentiable evolution and jumps.



(d) Differentiable evolution and jumps working against each other, jumps dominating.

Figure 10.7: When many function like the one shown in Figure 10.5 are averaged, the large-scale behaviour of the mean (dashed) might be dominated by the differentiable evolution as in (a), or by the jumps as in (b). The large-scale behaviour could also be a “convex combination” of both as in (c), or result from differentiable evolution and jumps working against each other as in (d) .

is run many times (with different seeds), and results are averaged in order to estimate an expected value of some output variable; e. g., an expected energy deposition in a layer of a calorimeter. When AD comes into the game, users would likely be interested in the derivative of such an (estimated) expected value.

In line with the AD theory (Chapter 2) and also as we checked in the previous Section 10.3 for a single MC iteration, AD computes the local derivative of the differentiable evolution, and is not concerned with the large-scale evolution including nearby jumps. When many graphs like Figure 10.5 are averaged, there are two basic ways how they could combine:

- The large-scale behaviour of the mean could be dominated by the differentiable evolution, with jumps creating some noise but largely cancelling out, as illustrated in Figure 10.7a. In this case, the algorithmic derivative of the mean, i. e. the local slope that we can compute with AD, would be close to the derivative of the large-scale behaviour of the mean, i. e. the slope that we are interested in.

- Opposite to this, the differentiable evolution could cancel out and the large-scale change of mean energy deposition could result solely from the jumps, e. g. as illustrated in Figure 10.7b. In this case, the algorithmic derivative of the mean would be close to zero, and thus unable to reflect the large-scale slope.

Of course, one might also observe a mixture of these two ways as in Figure 10.7c. There are even worse combinations like a mean differentiable evolution completely opposite to the large-scale trend but jumps overcompensating for it, as in Figure 10.7d. In the latter example, the algorithmic derivative of the mean would be negative even though the large-scale slope would be positive.

10.4.b. Derivatives of expected values vs. expected pathwise derivatives. To put these thoughts into stochastic terms, let us add an argument ω of a probability space Ω to the function f representing a single run of the Monte-Carlo iteration:

$$\begin{aligned} f : \mathbb{R} \times \Omega &\to \mathbb{R} \\ (x, \omega) &\mapsto f(x, \omega). \end{aligned} \tag{10.2}$$

If AD is used without special attention to randomness, it treats random variables much like constants, and computes the *pathwise derivative* $\frac{\partial}{\partial x} f(x, \omega)$, which itself is a random variable. Computing the mean of many realization of that random variable is an estimator for its expected value

$$\mathbb{E}_\omega \left[\frac{\partial f}{\partial x}(x, \omega) \right]. \tag{10.3}$$

What we want to compute, though, is the derivative of the expected value of f,

$$\frac{\partial}{\partial x} \left[\mathbb{E}_\omega f(x, \omega) \right], \tag{10.4}$$

as those would be required for e. g. an optimization of $\mathbb{E}_\omega f(x, \omega)$, or of an objective function depending on it. Equality of (10.3) and (10.4) holds e. g. under the assumption that $\frac{\partial}{\partial x} f(x, \omega)$ exists for all x and ω and is uniformly bounded by an integrable random variable $L(\omega)$, see Voss[183]. On the other hand, it is easy to construct counter-examples if jumps are allowed.[17] As even simple particle transport simulators contain jumps,[105] we cannot expect that (10.3) and (10.4) are equal. In other words, means of pathwise derivatives are only a *biased* estimator of (10.4). Techniques like the *reparametrization trick*[109], the *likelihood ratio* or *score function* method[71,165], and the *stochastic AD* method by Arya et al.[17] have been proposed to create unbiased estimators even when there are jumps; however, to our knowledge, they all come with assumptions and are not generally applicable to arbitrary code.

10.4.c. Test Setup. We therefore first took a look at how much (10.3) and (10.4) actually differ for a simple Geant4-like high energy physics (HEP) simulation. Nearly equally important, we had to see how large the variance of pathwise algorithmic derivatives would be – the larger it is, the more samples are required to make their mean a reliable estimate of their expected value (10.3).

Specifically, we have studied these questions for a Monte-Carlo simulation of electromagnetic showers in a simple sampling calorimeter with 50 pairs of an absorber layer made from lead tungstate and a gap layer filled with liquid argon,[8] similar to the TestEm3 example of Geant4. The simulation was conducted using the toolkit G4HepEm[142] and the application HepEmShow[141], which model all the relevant electromagnetic physics processes but consider only a very simple geometric setup. Pathwise derivatives have been computed using the AD tool CoDiPack[158] after first experiments with Derivgrind have shown promising results.

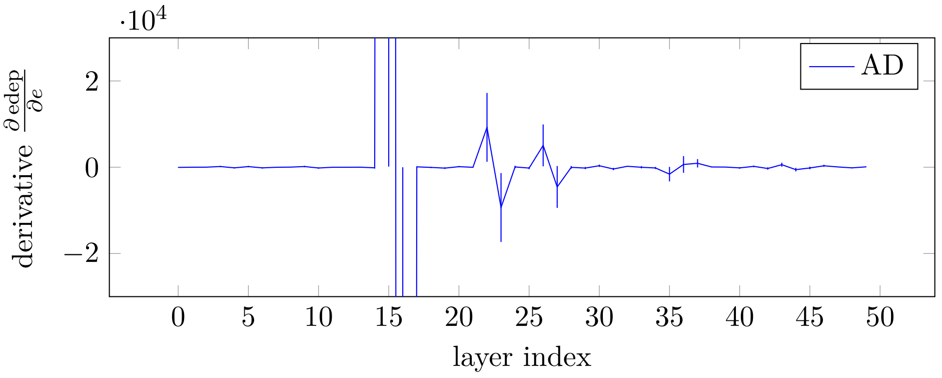
10.4.d. With all physics processes, noisy algorithmic derivatives. Figure 10.8 shows the average pathwise derivatives of the energy deposition in the fifty calorimeter layers with respect to the energy of the incoming primary particles (blue). For comparison, difference quotients approximating (10.4) are shown (black). In Figure 10.8a, we see that without any changes to G4HepEm/HepEmShow beyond what is needed to integrate CoDiPack, the pathwise derivatives have an extremely large variance. Thus, we cannot approximate (10.3) with a computationally feasible number N of simulated events.

10.4.e. After disabling multiple scattering, good agreement. When one particular physics process called *multiple scattering* (MSC) is deactivated, the variance of the pathwise derivatives is greatly reduced, and their expected value (10.3) comes quite close to the actual derivative (10.4) approximated with a central difference quotient, as shown in Figure 10.8b.

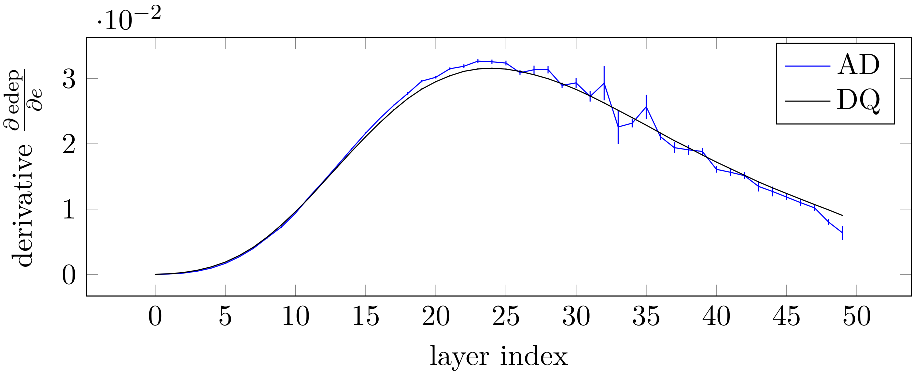
To see what the difference is, we can look at plots that show the mean energy deposition in one particular layer as a function of the primary energy, varying around 10 GeV (analogous to Figure 10.5). While the energy deposition computed with and without MSC (and another process called *energy loss fluctuations* included for the purpose of presentation) look very similar on a large scale plot in Figure 10.9, the zoomed plot in Figure 10.10 shows a clear qualitative difference: With MSC (and fluctuations) the function is very noisy, while without, it is piecewise differentiable and the slope of the line segments approximately match the large-scale derivative visible from Figure 10.9, similar to Figure 10.7a.

Coming back to Figure 10.8, to confirm our results, we produced Figure 10.8c with much more events and a much smaller interval for the difference quotients, to reduce stochastic and numerical errors. A deviation between (10.3) and (10.4) of about 5 % in the central layers remains. This is much less than we expected in the beginning.

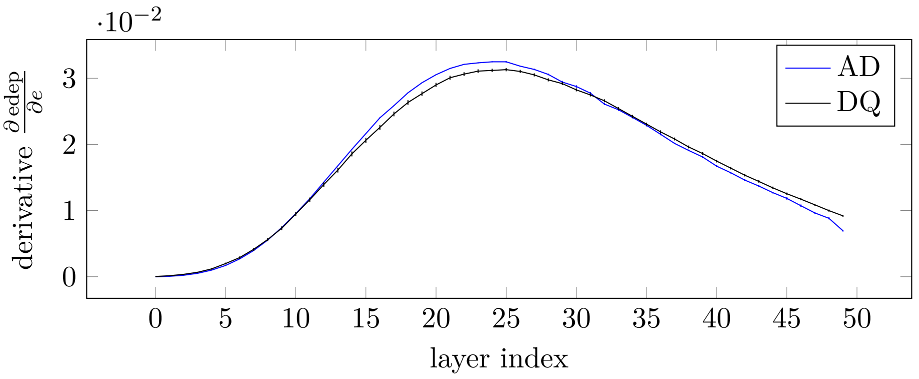
Similar statements hold for the derivatives of the energy deposition with respect to the geometrical thicknesses of the absorber and gap layers of the sampling calorimeter, as shown in Figure 10.11.



(a) Default configuration of G4HepEm with all physics processes, 24 M events.



(b) All physics processes except for multiple scattering, 24 M events. Difference quotients over 9.9. . . 10.1 GeV.



(c) 864 M events and smaller interval 9.995. . . 10.005 GeV for the difference quotient.

Figure 10.8: Algorithmic derivative of the edep in the calorimeter layers with respect to the primary energy e.

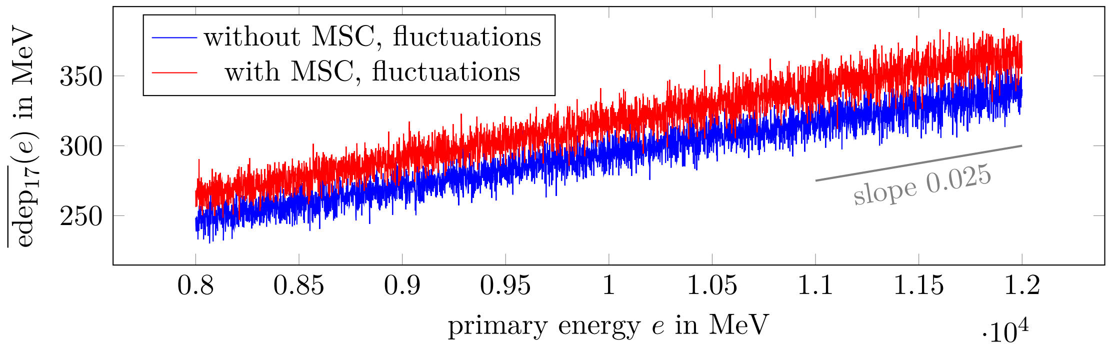


Figure 10.9: Dependency of the simulated mean energy deposition in layer 17 on the primary energy e. For every point in this plot, $N = 100$ events were simulated using the same random seed. Figure 10.10 zooms into this plot to see if these "noisy" functions are differentiable and if their derivatives match the large-scale slope of 0.025.

10.4.0.0.1. Using derivatives for optimization. As the derivative (10.4) is not a physical measurement but merely a tool to steer the gradient-based optimization algorithm into good descent directions, the bias of 5 %, in addition to stochastic errors, modelling errors, and errors due to disabling multiple scattering, can be perfectly acceptable. To demonstrate this, we have used our mean pathwise derivative estimator to minimize a simple loss function in order to reconstruct the primary energy e and the absorber thickness a from a given energy deposition distribution in the fifty calorimeter layers. With fixed step-sizes of 1 for e and $10^{-7}\,\text{mm}^2\,\text{MeV}^{-2}$ for a (accounting for their different units and orders of magnitude), and 1 k event simulated in each step for 350 steps, the stochastic gradient descent optimizer robustly converges from an initial solution $e^{(0)} = 22\,\text{GeV}$ and $a^{(0)} = 3\,\text{mm}$ to the correct minimizer, as shown in Figure 10.12.

10.4.0.0.2. Outlook: Going back to Geant4. We have also taken already a few steps to bring these encouraging results, obtained for the G4HepEm/HepEmShow package, to the scale of Geant4. Specifically, we used Derivgrind to evaluate derivatives of a Geant4 simulation of the same calorimeter geometry, available in the examples directory of G4HepEm, using either

- the G4HepEm toolkit to model electromagnetic physics in Geant4, with MSC disabled; or, in a second step,
- the native Geant4 electromagnetic processes except for MSC.

It was more or less straightforward to compute these derivatives with Derivgrind. Figure 10.13 shows that they reproduce our findings from G4HepEm/HepEmShow; the high variance comes from the fact that we averaged over much less events (64 000 instead of millions), given that Derivgrind does not target high-performance AD computations. Thus, neither the more general and more complex geometry handling of Geant4, nor the

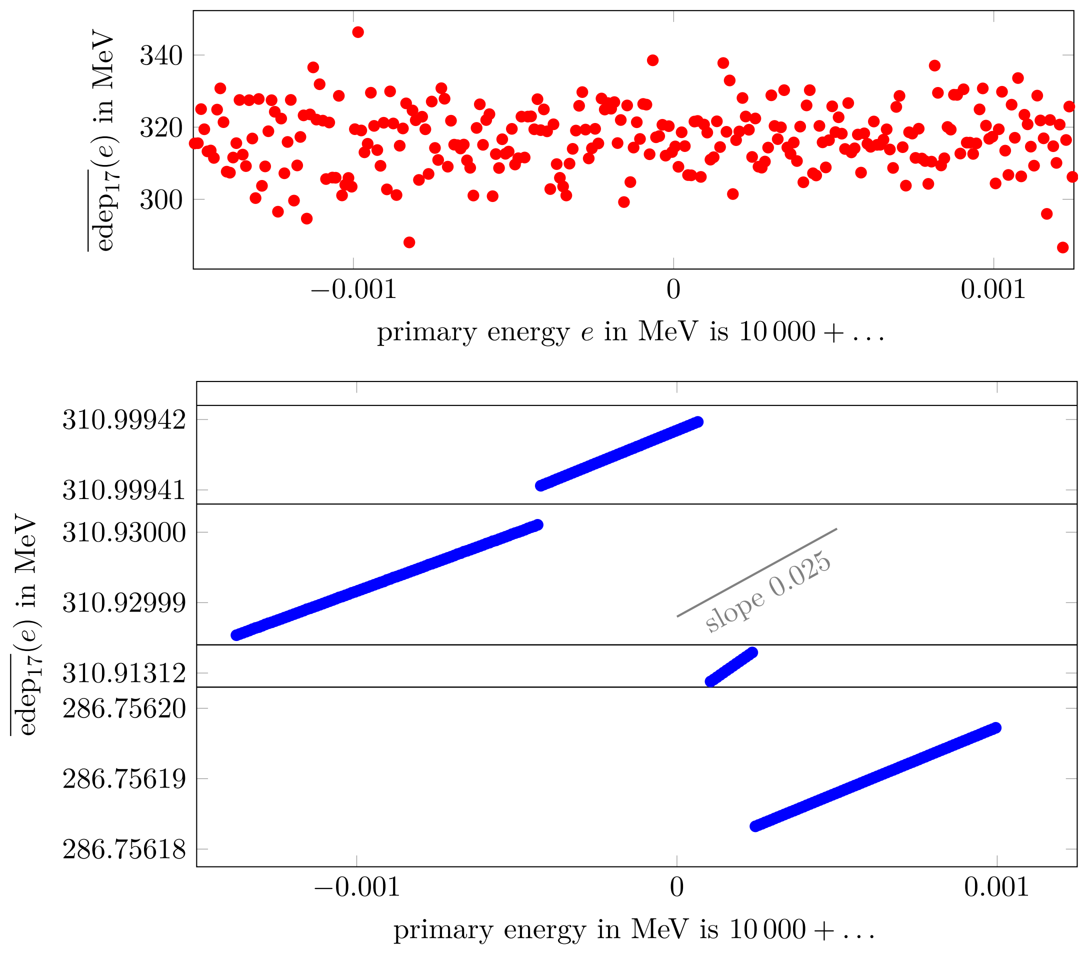


Figure 10.10: Zoom into figure 10.9, showing the mean energy deposition in layer 17 plotted over a much smaller range of the primary energy e. Again, each point represents a HepEmShow simulation of $N = 100$ events, always using the same random seed. The energy deposition computed with the full set of physics processes still appears noisy (top). With multiple scattering and energy loss fluctuations disabled, however, the averaged energy deposition is a piecewise differentiable function of the primary energy, and its derivative (i. e. the slope of the segments) approximately matches the large-scale slope observed in Figure 10.9.

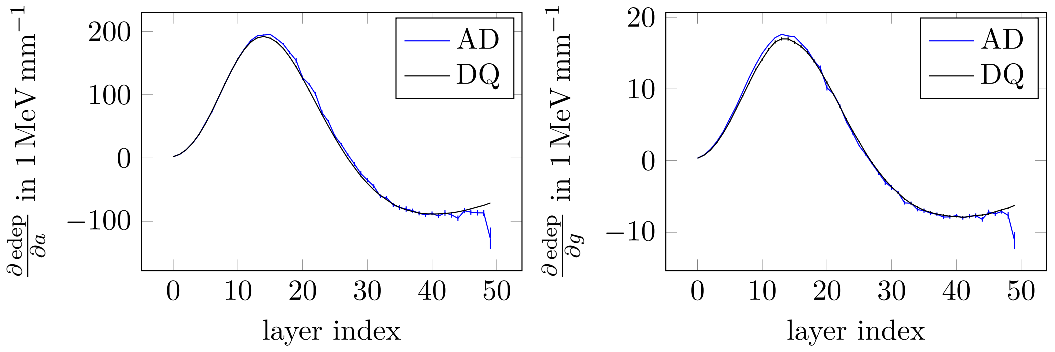


Figure 10.11: Algorithmic derivative of the energy deposition with respect to the absorber thickness a (left) and gap thickness g (right).

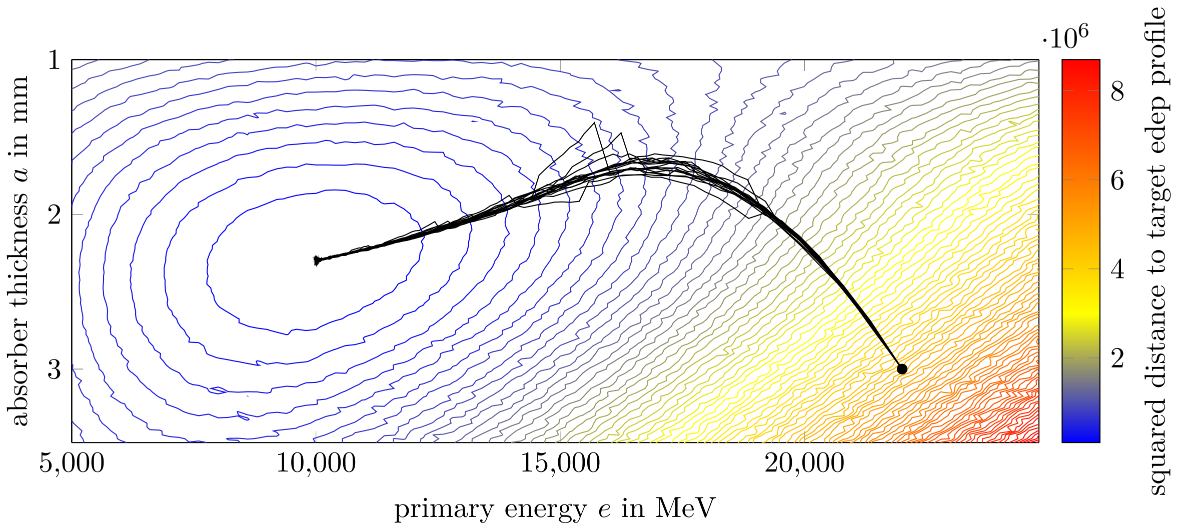


Figure 10.12: Reconstruction of the values of primary energy and absorber thickness that lead to a given energy deposition profile in a sampling calorimeter, using the (stochastic) gradient descent optimizer with pathwise algorithmic derivatives of the shower simulation. All 16 computed trajectories robustly converge to the correct minimizer.

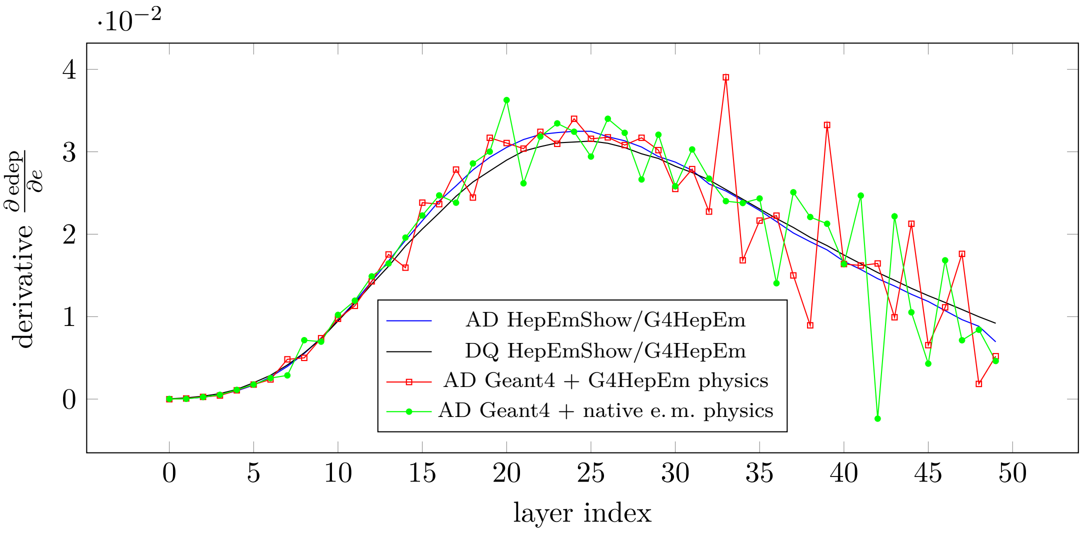


Figure 10.13: Derivgrind derivatives of energy depositions computed by Geant4, using the G4HepEm physics (red) or native electromagnetic physics list (green), both with MSC disabled. Only 64 000 events have been simulated for this exploratory study, which is why the variance is comparatively large. The results seem to approximate the algorithmic (blue) and numeric derivatives (black) of the HepEmShow/G4HepEm package already shown in Figure 10.8c.

different implementation of the electromagnetic processes in Geant4, introduce mathematical problems for AD. Our exploratory study with Derivgrind thus motivates and justifies investments in bringing source-code-based AD into Geant4.

11. Conclusion

11.1. Summary

In this dissertation, we have developed the machine-code-based AD tool *Derivgrind*. Starting with an introduction to algorithmic differentiation (Chapter 2) and dynamic binary instrumentation (Chapter 4) of machine code (Chapter 3), we have developed the key ideas and steps to leverage the Valgrind framework to augment compiled software with AD logic for the forward mode (Chapter 5), for tape recording passes enabling the reverse mode of AD (Chapter 6), and for the heuristic detection of bit-tricks (Chapter 7).

We have validated Derivgrind's results for many small test programs, the Python interpreter CPython and the spreadsheet program LibreOffice Calc (Chapter 8), as well as for medical imaging and high-energy physics software based on the Monte-Carlo particle simulation toolkit Geant4 (Chapter 10). This wide variety of tests demonstrate that machine-code-based AD can provide accurate derivatives for complex, cross-language and possibly partially closed-source software projects, and that only little manual efforts are usually required for the AD setup. Among the wide spectrum of available AD implementations, Derivgrind is therefore well-suited for exploratory studies of AD in new application domains and for productive use in setups where source-code-based AD is not yet feasible.

In general, machine-code-based AD is more prone than source-code based AD to silently computing wrong derivatives due to bit-tricks in the program to be differentiated (Section 3.3). With Derivgrind, we observed that this was only a mild limitation for most of the aforementioned application examples, and we also provide a heuristic bit-trick-finder instrumentation. It remains to be seen how much future machine-code-based AD tools are able to improve Derivgrind's run-time performance, which could not catch up with source-code-based AD tools in our performance measurements (Chapter 9).

Nevertheless, already today, Derivgrind has powered exploratory studies of AD in high-energy physics that would not have been possible without it.

11.2. Outlook

There is a lot more to be discovered about machine-code-based AD, and we would like to highlight the following directions as particularly interesting for further research and development.

11.2.a. More Applications. While Derivgrind has been applied to a variety of open-source software projects in this thesis, it would be interesting to see how well it performs in an industrial context with proprietary software. Such studies might add more bit-tricks to our list in Section 3.3.

11.2.b. More AD Features. This thesis follows a single way of implementing forward- and reverse-mode AD, respectively. Depending on the application, it might be worthwhile to implement more advanced AD features, like more efficient tape layouts, preaccumulation, or higher-order derivatives.

11.2.c. More Platforms. Derivgrind supports the x86-64 Linux platform, including vector extensions up to AVX2. Also, a lot of 32-bit x86 Linux is supported. The Valgrind framework is able to run on a few more instruction set architectures such as ARM64 and ppc64, and there is some work going on regarding RISC-V. It may be interesting to test Derivgrind on these platforms and to add any missing support. In particular, this would allow to find out about the prevalence and types of bit-tricks in compiled programs on platforms other than x86-64.

11.2.d. Other Applications for Dynamic Binary Instrumentation of Floating-Point Calculations. While DBI tools are typically used for type-agnostic purposes like debugging, profiling or security reseach, a few DBI tools listed in Paragraph 4.1.3.f allow to analyze and improve the accuracy of floating-point calculations. Besides this well-established purpose and our new AD instrumentation, we can imagine other interesting types of analyses of floating-point calculations in computer programs.

For example, in parallel programs, non-deterministic thread or process scheduling can affect the order in which real-arithmetic operations are performed. As floating-point arithmetic is not associative, this might result in non-deterministic floating-point errors. A DBI tool could be developed to check that in several runs of the client program, evaluations are always algebraically equivalent. One idea for that would be to shadow the basic real-arithmetic operations by equivalents in a finite field $\mathbb{F}_{p^k}$, where they can be implemented in an exact fashion.

11.2.e. Better Performance. We paid attention to performance while developing Derivgrind, but it has not been our main objective. It would be interesting to see if the performance can be improved, e. g. by using other DBI frameworks.

11.2.f. Static Binary Instrumentation. As a dynamic binary instrumentation tool, Derivgrind operates on the machine code shortly before it has a chance to execute on the CPU; the differentiated machine code only exists in RAM. One may instead try a static instrumentation approach, similar to the proof-of-concept forward-mode AD tool prototype adac by Gendler et al.[69], to produce a differentiated executable file. Intuitively, static instrumentation can operate on the entire program at once, which may allow to switch from a tape-recording to a source-rewriting reverse mode (Section 2.4.4), and enable more code optimizations. On the other hand, static instrumentation does not have access to run-time information and may struggle with dynamic loading, self-modifying code etc. Last but not least, different tooling would be required.

11.2.g. Differentiation at Even Lower Levels. As mentioned in Section 4.1.1, besides dynamic (and static) binary instrumentation, another approach to differentiate machine code is to design differentiated hardware, as done by Schoder,[164] or to implement differentiated emulators. These approaches would allow to differentiate through kernel code as well, so the content of files could be declared as AD input or output and MPI parallelism on a single host would be naturally supported as well.

A. Bit-Tricks in Well-Known Software Packages

A.1. Natural Logarithm in Geant4

Geant4 implements functions `G4LogConsts::getMantExponent` and `G4Log` that behave similarly to the `math.h` functions `frexp` and `log`, respectively.

`frexp` scales a floating-point number by an integer power of two such that it ends up between $\frac{1}{2}$ and 1 (or $-\frac{1}{2}$ and -1). `getMantExponent` achieves this by the bit-trick described in Paragraph 3.3.4.a, overwriting the eleven exponent bits by `0b01111111110` as shown in Listing A.1.

To give a few more details on the implementation of `G4Log` in Listing A.2, note that it is based on a rational function approximation

$$\frac{1.02 \times 10^{-4} \cdot x^5 + 0.497 \cdot x^4 + 4.71 \cdot x^3 + 14.5 \cdot x^2 + 17.9 \cdot x + 7.71}{x^5 + 11.3 \cdot x^4 + 45.2 \cdot x^3 + 83.0 \cdot x^2 + 71.2 \cdot x + 23.1} \cdot x^3 - \frac{x^2}{2} + x \quad (A.1)$$

for $\ln(1 + x)$; coefficients in equation (A.1) have been rounded for the purpose of presentation. As the approximation is good in a neighborhood of $x = 0$ only, the argument `x` is first mapped between $\sqrt{\frac{1}{2}}$ and $\sqrt{2}$. To this end, `getMantExponent` is followed by a conditional multiplication with two if the result is less or equal to `SQRTH` $= \sqrt{\frac{1}{2}}$. See Listing A.2 for details. The functions `get_log_px` and `get_log_qx` evaluate the numerator and denominator of the big fraction in (A.1).

A.2. Math Functions for 32-Bit Numbers in NumPy

On our test system, NumPy (in version 1.19.5) implements a few math functions on its own for 32-bit components of SIMD vectors. In this section, we provide a more detailed analysis of our observations reported in Paragraph 8.2.d about the `exp` function.

A.2.a. Implementation of `exp`. Listing A.3 is an extract from the template for a code generation tool producing the source code of NumPy version 1.19.5. Symbols enclosed by `@` signs are replaced by the code generation tool before compilation. The snippet highlights that NumPy uses a rational function approximation, and, like Geant4's `G4Log` in Appendix A.1, maps the argument into an interval where the approximation is most accurate. To that end, for each component, the code rounds the product of the argument with `log2e` $= \frac{1}{\ln 2}$ to an integer k, using a bit-trick (Paragraph A.2.b), and

Listing A.1: Implementation of an alternative `frexp` function `getMantExponent`, copied from `G4Log.hh` in Geant4 tag v11.0.0. The two helper functions were copied from the same file.

```
inline G4double getMantExponent(const G4double x, G4double& fe)
{
  uint64_t n = dp2uint64(x);

  // Shift to the right up to the beginning of the exponent.
  // Then with a mask, cut off the sign bit
  uint64_t le = (n >> 52);

  // chop the head of the number: an int contains more than 11 bits (32)
  int32_t e =
    le;  // This is important since sums on uint64_t do not vectorise
  fe = e - 1023;

  // This puts to 11 zeroes the exponent
  n &= 0x800FFFFFFFFFFFFFULL;
  // build a mask which is 0.5, i.e. an exponent equal to 1022
  // which means *2, see the above +1.
  const uint64_t p05 = 0x3FE0000000000000ULL;  // dp2uint64(0.5);
  n |= p05;

  return uint642dp(n);
}
// helper functions:
inline uint64_t dp2uint64(G4double x)
{
  ieee754 tmp;
  tmp.d = x;
  return tmp.ll;
}
inline G4double uint642dp(uint64_t ll)
{
  ieee754 tmp;
  tmp.ll = ll;
  return tmp.d;
}
```

Listing A.2: Implementation of an alternative natural logarithm function `G4Log`, copied from `G4Log.hh` in Geant4 tag v11.0.0.

```
inline G4double G4Log(G4double x)
{
  const G4double original_x = x;

  /* separate mantissa from exponent */
  G4double fe;
  x = G4LogConsts::getMantExponent(x, fe);

  // blending
  x > G4LogConsts::SQRTH ? fe += 1. : x += x;
  x -= 1.0;

  /* rational form */
  G4double px = G4LogConsts::get_log_px(x);

  // for the final formula
  const G4double x2 = x * x;
  px *= x;
  px *= x2;

  const G4double qx = G4LogConsts::get_log_qx(x);

  G4double res = px / qx;

  res -= fe * 2.121944400546905827679e-4;
  res -= 0.5 * x2;

  res = x + res;
  res += fe * 0.693359375;

  if(original_x > G4LogConsts::LOG_UPPER_LIMIT)
    res = std::numeric_limits<G4double>::infinity();
  if(original_x < G4LogConsts::LOG_LOWER_LIMIT)  // THIS IS NAN!
    res = -std::numeric_limits<G4double>::quiet_NaN();

  return res;
}
```

`@isa@_range_reduction` in Listing A.4 subtracts $k \cdot \ln 2$ from the argument; note that `codyw_c1`, `codyw_c2` and `zeros_f` sum up to $(-\ln 2)$. Correspondingly, in the end the result is scaled by 2^k, using `@isa@_scalef_ps` from Listing A.5 which contains another bit-trick as described further down in Paragraph A.2.c.

A.2.b. Rounding bit-trick. As highlighted in Listing A.3, the rounding is done by adding and subtracting `cvt_magic` $= 1.5 \cdot 2^{23}$, explointing floating-point inaccuracies as described in Section 3.3.5.

A.2.c. Bit-trick for `ldexp`. NumPy's `fma_scalef_ps`, called from Listing A.3 and shown in Listing A.5, is a SIMD version of the `ldexpf` function that takes a `binary32` argument `arg` and an integer argument `exp`, and returns `arg` $\cdot\, 2^{\texttt{exp}}$. The code uses the bit-trick described in Paragraph 3.3.4.b, as explained in the following. We focus on the `else` branch, which deals with arguments that are normal numbers. The intrinsic function `_mm256_slli_epi32` normally resolves to an AVX2 instruction that performs a bitshift to the left on each 32-bit SIMD component. As `binary32` has 23 significand bits, this aligns the integer `exponent` with the `binary32` exponent bits in each 32-bit SIMD component. The `_mm256_add_epi32` intrinsic performs an integer addition, and `_mm256_castps_si256` and `_mm256_castsi256_ps` cast between integer and floating-point types.

A.2.d. Eliminating bit-tricks via build configuation. When NumPy is configured with `--cpu-baseline="SSE"` and `--cpu-dispatch="SSE"` to turn off AVX2 instructions, a different codepath without unsupported bit-tricks is used to compute the exponential function, as already reported in Paragraph 8.2.d.

A.3. Rounding via Floating-Point Inaccuracies in the GLIBC Math Library

The function `reduce_sincos` in Listing A.6, copied from the GLIBC source file

`sysdeps/ieee754/dbl-64/s_sin.c`

at version `glibc-2.35`, is used to shift the argument of sin into the interval between $-\frac{\pi}{4}$ and $\frac{\pi}{4}$ by adding a multiple of $\frac{\pi}{2}$. The constant `hpinv` and `toint` is $1.5 \cdot 2^{52}$. Adding `toint`, and subtracting it right away, introduces floating-point inaccuracies that round a number to the next integer, as discussed in Section 3.3.5. `mp1` and `mp2` add up to $\frac{\pi}{2}$, so `y` basically contains the reduced argument. The remainder of the function improves accuracy, but the changes are minimal as `pp3` is about -4.98×10^{-17} and `pp4` is about -1.90×10^{-25}.

The actual `__sin` function in Listing A.7 distinguishes several cases depending on the magnitude of the argument. The above `reduce_sincos` function is used between 2.42 and 1.05×10^{8}. Between 2^{-26} and 0.855, the look-up table based approach of `do_sin`

Listing A.3: Partial implementation of a SIMD function behaving like `expf` on each 32-bit component of a SIMD vector, adapted from `simd.inc.src` in NumPy version v1.19.5. The use of a bit-trick for rounding (Paragraph A.2.b) is highlighted.

```
static NPY_GCC_OPT_3 NPY_GCC_TARGET_@ISA@ void
@ISA@_exp_FLOAT(npy_float * op,
                npy_float * ip,
                const npy_intp array_size,
                const npy_intp steps)
{

    /* ... load constants, stride, number of lanes, ... */


    while (num_remaining_elements > 0) {

        /* ... load x, identify very small and large components ...*/

        quadrant = _mm@vsize@_mul_ps(x, log2e);

        /* round to nearest */
        quadrant = _mm@vsize@_add_ps(quadrant, cvt_magic);
        quadrant = _mm@vsize@_sub_ps(quadrant, cvt_magic);

        /* Cody-Waite's range reduction algorithm */
        x = @isa@_range_reduction(x, quadrant, codyw_c1, codyw_c2, ↩
            zeros_f);

        num_poly = @fmadd@(exp_p5, x, exp_p4);
        num_poly = @fmadd@(num_poly, x, exp_p3);
        num_poly = @fmadd@(num_poly, x, exp_p2);
        num_poly = @fmadd@(num_poly, x, exp_p1);
        num_poly = @fmadd@(num_poly, x, exp_p0);
        denom_poly = @fmadd@(exp_q2, x, exp_q1);
        denom_poly = @fmadd@(denom_poly, x, exp_q0);
        poly = _mm@vsize@_div_ps(num_poly, denom_poly);

        /*
         * compute val = poly * 2^quadrant; which is same as adding the
         * exponent of quadrant to the exponent of poly. quadrant is an↩
             int,
         * so extracting exponent is simply extracting 8 bits.
         */
        poly = @isa@_scalef_ps(poly, quadrant);

        /* ... mask results for small and large arguments with 0 and ↩
            inf,
         *     respectively, and store poly ... */
    }

    if (@mask_to_int@(overflow_mask)) {
        npy_set_floatstatus_overflow();
    }
}
```

Listing A.4: SIMD implementation of Cody-Waite's range reduction algorithm, copied from `simd.inc.src` in NumPy version v1.19.5. The function computes $\mathtt{x} + \mathtt{y} \cdot (\mathtt{c1} + \mathtt{c2} + \mathtt{c3})$.

```
static NPY_INLINE NPY_GCC_OPT_3 NPY_GCC_TARGET_@ISA@ @vtype@
@isa@_range_reduction(@vtype@ x, @vtype@ y, @vtype@ c1, @vtype@ c2, ↩
    @vtype@ c3)
{
    @vtype@ reduced_x = @fmadd@(y, c1, x);
    reduced_x = @fmadd@(y, c2, reduced_x);
    reduced_x = @fmadd@(y, c3, reduced_x);
    return reduced_x;
}
```

Listing A.5: Implementation of a SIMD function behaving like `ldexpf` on each 32-bit component of a 256-bit SIMD vector, copied from `simd.inc.src` in NumPy version v1.19.5. The highlighted lines perform an integer addition to the exponent bits (Paragraph 3.3.4.b).

```
static NPY_INLINE NPY_GCC_OPT_3 NPY_GCC_TARGET_AVX2 __m256
fma_scalef_ps(__m256 poly, __m256 quadrant)
{
    /*
     * Handle denormals (which occur when quadrant <= -125):
     * 1) This function computes poly*(2^quad) by adding the exponent of
     poly to quad
     * 2) When quad <= -125, the output is a denormal and the above logic
     breaks down
     * 3) To handle such cases, we split quadrant: -125 + (quadrant + 125)
     * 4) poly*(2^-125) is computed the usual way
     * 5) 2^(quad-125) can be computed by: 2 << abs(quad-125)
     * 6) The final div operation generates the denormal
     */
     __m256 minquadrant = _mm256_set1_ps(-125.0f);
     __m256 denormal_mask = _mm256_cmp_ps(quadrant, minquadrant, _CMP_LE_OQ);
     if (_mm256_movemask_ps(denormal_mask) != 0x0000) {
        __m256 quad_diff = _mm256_sub_ps(quadrant, minquadrant);
        quad_diff = _mm256_sub_ps(_mm256_setzero_ps(), quad_diff);
        quad_diff = _mm256_blendv_ps(_mm256_setzero_ps(), quad_diff, ↩
            denormal_mask);
        __m256i two_power_diff = _mm256_sllv_epi32(
                                   _mm256_set1_epi32(1), _mm256_cvtps_epi32(↩
                                       quad_diff));
        quadrant = _mm256_max_ps(quadrant, minquadrant); //keep quadrant >= -126
        __m256i exponent = _mm256_slli_epi32(_mm256_cvtps_epi32(quadrant), 23);
        poly = _mm256_castsi256_ps(
                   _mm256_add_epi32(
                       _mm256_castps_si256(poly), exponent));
        __m256 denorm_poly = _mm256_div_ps(poly, _mm256_cvtepi32_ps(↩
            two_power_diff));
        return _mm256_blendv_ps(poly, denorm_poly, denormal_mask);
     }
     else {
        __m256i exponent = _mm256_slli_epi32(_mm256_cvtps_epi32(quadrant), 23);
        poly = _mm256_castsi256_ps(
                   _mm256_add_epi32(
                       _mm256_castps_si256(poly), exponent));
        return poly;
     }
}
```

Listing A.6: Range reduction function in the GLIBC math library. The code has been copied from `sysdeps/ieee754/dbl-64/s_sin.c` in version `glibc-2.35`.

```
/* Reduce range of x to within PI/2 with abs (x) < 105414350.  The high part
   is written to *a, the low part to *da.  Range reduction is accurate to 136
   bits so that when x is large and *a very close to zero, all 53 bits of *a
   are correct.  */
static __always_inline int4
reduce_sincos (double x, double *a, double *da)
{
  mynumber v;

  double t = (x * hpinv + toint);
  double xn = t - toint;
  v.x = t;
  double y = (x - xn * mp1) - xn * mp2;
  int4 n = v.i[LOW_HALF] & 3;

  double b, db, t1, t2;
  t1 = xn * pp3;
  t2 = y - t1;
  db = (y - t2) - t1;

  t1 = xn * pp4;
  b = t2 - t1;
  db += (t2 - b) - t1;

  *a = b;
  *da = db;
  return n;
}
```

in Listing A.8 is used. In this function, the constant `big` $= 1.5 \cdot 2^{45}$ is added to the argument, rounding it to an integer multiple `u.x` of 2^{-7}. In the following, we refer to the argument of the sine function as x, to its closest integer multiple of 2^{-7} as x_{tab} and to the difference as $x_{\text{rem}} = x - x_{\text{tab}}$. `SINCOS_TABLE_LOOKUP` uses the last-significant bits of `u.x` as indices into a look-up table, reading `sn` $= \sin(x_{\text{tab}})$, `cs` $= \cos(x_{\text{tab}})$ and further small terms `ssn` and `ccs`. Furthermore, the code finds `s` $= \sin(x_{\text{rem}})$ and `c` $= 1 - \cos(x_{\text{rem}})$ via Taylor expansions. The return values combines these results, mainly according to the trigonometric formula

$$\sin(x) = \sin(x_{\text{tab}} + x_{\text{rem}}) = \sin(x_{\text{tab}})\cos(x_{\text{rem}}) + \cos(x_{\text{tab}})\sin(x_{\text{rem}}),$$

plus some numerical adjustments using `ssn` and `ccs` and a proper choice of the sign.

Listing A.7: Sine function in the GLIBC math library. The code has been copied from `sysdeps/ieee754/dbl-64/s_sin.c` in version `glibc-2.35`.

```c
/*******************************************************************/
/* An ultimate sin routine. Given an IEEE double machine number x   */
/* it computes the rounded value of sin(x).         */
/*******************************************************************/
#ifndef IN_SINCOS
double
SECTION
__sin (double x)
{
  double t, a, da;
  mynumber u;
  int4 k, m, n;
  double retval = 0;

  SET_RESTORE_ROUND_53BIT (FE_TONEAREST);

  u.x = x;
  m = u.i[HIGH_HALF];
  k = 0x7fffffff & m;   /* no sign            */
  if (k < 0x3e500000)   /* if x->0 =>sin(x)=x */
    {
      math_check_force_underflow (x);
      retval = x;
    }
/*--------------------------- 2^-26<|x|< 0.855469---------------------- */
  else if (k < 0x3feb6000)
    {
      /* Max ULP is 0.548.   */
      retval = do_sin (x, 0);
    }       /*   else  if (k < 0x3feb6000)    */

/*---------------------------- 0.855469  <|x|<2.426265  ----------------------*/
  else if (k < 0x400368fd)
    {
      t = hp0 - fabs (x);
      /* Max ULP is 0.51.   */
      retval = copysign (do_cos (t, hp1), x);
    }       /*   else  if (k < 0x400368fd)    */

/*--------------------------- 2.426265<|x|< 105414350 ----------------------*/
  else if (k < 0x419921FB)
    {
      n = reduce_sincos (x, &a, &da);
      retval = do_sincos (a, da, n);
    }       /*   else  if (k <  0x419921FB )    */

/* --------------------105414350 <|x| <2^1024------------------------------*/
  else if (k < 0x7ff00000)
    {
      n = __branred (x, &a, &da);
      retval = do_sincos (a, da, n);
    }
/*--------------------- |x| > 2^1024 ----------------------------------*/
  else
    {
      if (k == 0x7ff00000 && u.i[LOW_HALF] == 0)
  __set_errno (EDOM);
      retval = x / x;
    }

  return retval;
}
```

Listing A.8: Sine function in the GLIBC math library. The code has been copied from `sysdeps/ieee754/dbl-64/s_sin.c` in version `glibc-2.35`.

```c
/* Given a number partitioned into X and DX, this function computes the sine of
   the number by combining the sin and cos of X (as computed by a variation of
   the Taylor series) with the values looked up from the sin/cos table to get
   the result.  */
static __always_inline double
do_sin (double x, double dx)
{
  double xold = x;
  /* Max ULP is 0.501 if |x| < 0.126, otherwise ULP is 0.518.  */
  if (fabs (x) < 0.126)
    return TAYLOR_SIN (x * x, x, dx);

  mynumber u;

  if (x <= 0)
    dx = -dx;
  u.x = big + fabs (x);
  x = fabs (x) - (u.x - big);

  double xx, s, sn, ssn, c, cs, ccs, cor;
  xx = x * x;
  s = x + (dx + x * xx * (sn3 + xx * sn5));
  c = x * dx + xx * (cs2 + xx * (cs4 + xx * cs6));
  SINCOS_TABLE_LOOKUP (u, sn, ssn, cs, ccs);
  cor = (ssn + s * ccs - sn * c) + cs * s;
  return copysign (sn + cor, xold);
}
```

B. Further Code Snippets

B.1. Library Caller Details

AD inputs and outputs are not necessarily specified by variable names and line numbers in the source code. This section deals with the case of a compiled function in a shared object with a known signature that comprises all AD inputs and outputs. Specifically, we assume that the signature is

`void` ⟨*functionname*⟩ `(int, char*, int,` ⟨*fptype*⟩ `const*, int,` ⟨*fptype*⟩ `*)` (B.1)

with *fptype* denoting either `double` or `float`. The three pairs of an integer and a pointer specify three buffers: The first buffer contains parameters, i. e. data irrelevant for AD, and the second and third buffer contain AD inputs and outputs, respectively.

Our ML framework wrappers (Section 6.5.2) allow to embed functions with this signature into the AD workflow of PyTorch[146] and TensorFlow[1]. They make use of the library caller program shown in Listing B.1. When it is started via

`derivgrind-library-caller` ⟨*library.so*⟩ ⟨*functionname*⟩ ⟨*fptype*⟩
⟨*nParam*⟩ ⟨*nInput*⟩ ⟨*nOutput*⟩ ⟨*path*⟩, (B.2)

it dynamically loads the specified library (with `dlopen`) and retrieves a pointer to the specified function (with `dlsym`). The three buffers are allocated according to the specified size *nParam* and counts *nInput* and *nOutput*, and the parameters and AD input buffer are initialized with the content of the files `dg-libcaller-params`, `dg-libcaller-inputs` in the specified *path*. After declaring the AD inputs with the client request macro `DG_INPUTF`, the specified function is called. Finally, the AD outputs are declared with `DG_OUTPUTF`, and their values are written to the file `dg-libcaller-outputs` in the specified *path*.

If the signature of a function in a shared object is different from (B.1), the code in Listing B.1 would need to be adapted.

B.2. Function Wrapping Details

B.2.a. Some Valgrind Internals. As outlined in Section 4.3.3, in order to call the unwrapped function from the respective function wrapper, a macro like `CALL_FN_W_WW` has to be used as shown in Listing 4.6. Listing B.2 reproduces the definition of this macro for the x86-64 ISA in the file `valgrind.h` of Valgrind version 3.18.1. `VALGRIND_ALIGN_STACK` aligns RSP to a multiple of 8 bytes, which is undone by `VALGRIND_RESTORE_STACK`.

Listing B.1: Client program to record the execution of a function of a shared library.

```cpp
#include <iostream>
#include <fstream>
#include <cstdio>
#include <dlfcn.h>
#include "derivgrind.h"

template<typename fptype>
int main_fp(int argc, char* argv[]){
  // load library and symbol
  void* loaded_lib = dlopen(argv[1], RTLD_NOW);
  if(!loaded_lib){
    std::cerr << "Error loading shared object '" << argv[1] << "':\n" << dlerror() << std::endl;
    exit(EXIT_FAILURE);
  }
  using fptr = void (*)(int,char*,int,fptype const*,int, fptype*);
  fptr loaded_fun = (fptr)dlsym(loaded_lib, argv[2]);
  if(!loaded_fun){
    std::cerr << "Error loading symbol '" << argv[2] <<"':\n" << dlerror() << std::endl;
    exit(EXIT_FAILURE);
  }

  // sizes of non-differentiable parameters, differentiable inputs, differentiable outputs
  long long param_size, input_count, output_count;
  try { // parse from command-line arguments
    param_size = std::stoll(argv[4]);
    input_count = std::stoll(argv[5]);
    output_count = std::stoll(argv[6]);
  } catch (std::invalid_argument const& ex) {
    std::cerr << "Invalid argument:\n" << ex.what() << std::endl;
    exit(EXIT_FAILURE);
  } catch (std::out_of_range const& ex) {
    std::cerr << "Argument out of range:\n" << ex.what() << std::endl;
    exit(EXIT_FAILURE);
  }

  // buffers for non-diff parameters, diff inputs, diff outputs
  char* param_buf;
  fptype *input_buf, *output_buf;

  // read content from files
  std::string path = argv[7];
  std::ifstream param_file(path+"/dg-libcaller-params", std::ios::binary);
  std::ifstream input_file(path+"/dg-libcaller-inputs", std::ios::binary);

  param_buf = new char[param_size+1]; // +1 to avoid allocations of length zero
  input_buf = new fptype[input_count+1];
  output_buf = new fptype[output_count+1];
  if(!param_buf || !input_buf || !output_buf){
    std::cerr << "Failure to allocate buffers." << std::endl;
    exit(EXIT_FAILURE);
  }

  param_file.read((char*)param_buf, param_size);
  input_file.read((char*)input_buf, input_count*sizeof(fptype));

  // register inputs
  for(unsigned long long i=0; i<input_count; i++){
    DG_INPUTF(input_buf[i]);
  }
  // call the function
  loaded_fun(param_size, param_buf, input_count, input_buf, output_count, output_buf);
  // register outputs
  for(unsigned long long i=0; i<output_count; i++){
    DG_OUTPUTF(output_buf[i]);
  }

  // write binary output
  std::ofstream output_file(path+"/dg-libcaller-outputs", std::ios::binary);
  output_file.write((char*)output_buf, sizeof(fptype)*output_count);

  return 0;
}


int main(int argc, char* argv[]){
  if(argc<4){
    std::cerr << "Error: Bad number of arguments." << std::endl;
    exit(EXIT_FAILURE);
  }
  if(argv[3][0]=='d') return main_fp<double>(argc,argv);
  else if(argv[3][0]=='f') return main_fp<float>(argc,argv);
  else {
    std::cerr << "Error: Bad floating point type specification '" << argv[3] << "'" << std::endl;
    exit(EXIT_FAILURE);
  }
}
```

Listing B.2: Definition of the macro `CALL_FN_W_WW` for the x86-64 ISA in the file `valgrind.h` of Valgrind version 3.18.1.

```
#define CALL_FN_W_WW(lval, orig, arg1,arg2)                              \
   do {                                                                  \
      volatile OrigFn        _orig = (orig);                             \
      volatile unsigned long _argvec[3];                                 \
      volatile unsigned long _res;                                       \
      _argvec[0] = (unsigned long)_orig.nraddr;                          \
      _argvec[1] = (unsigned long)(arg1);                                \
      _argvec[2] = (unsigned long)(arg2);                                \
      __asm__ volatile(                                                  \
         VALGRIND_CFI_PROLOGUE                                           \
         VALGRIND_ALIGN_STACK                                            \
         "subq $128,%%rsp\n\t"                                           \
         "movq 16(%%rax), %%rsi\n\t"                                     \
         "movq 8(%%rax), %%rdi\n\t"                                      \
         "movq (%%rax), %%rax\n\t"  /* target->%rax */                   \
         VALGRIND_CALL_NOREDIR_RAX                                       \
         VALGRIND_RESTORE_STACK                                          \
         VALGRIND_CFI_EPILOGUE                                           \
         : /*out*/   "=a" (_res)                                         \
         : /*in*/    "a" (&_argvec[0]) __FRAME_POINTER                   \
         : /*trash*/ "cc", "memory", __CALLER_SAVED_REGS, "r14", "r15"   \
      );                                                                 \
      lval = (__typeof__(lval)) _res;                                    \
   } while (0)
```

`VALGRIND_CALL_NOREDIR_RAX` expands to a special sequence of four `rolq` and one `xchgq` instruction similar to the one emitted for client requests (Listing 4.5). It has no effect on a normal x86-64 processor, but translates into VEX code that jumps to the address in RAX without interference from the function wrapping mechanism.

The jump target address and the two integer arguments are stored in the array `_argvec` defined in the C code; the specification `"a" (&_argvec[0])` in the input operands list of the `__asm__` syntax makes the compiler initialize RAX with the array address. Thus, the first two `movq` statements copy the two integer arguments into the registers RDI and RSI, where they belong according to the System-V ABI calling convention (see also Listing 3.1). After the call, the return value of integer type in RAX is made available in the C code via the `"=a" (_res)` output operand.

The macro `__CALLER_SAVED_REGS` expands to a list of registers that the original function may modify according to the calling conventions. In the clobber list of the extended `__asm__` syntax, it makes the compiler aware that the assembly code may change them.

B.2.b. Extensions of the Valgrind Framework. Derivgrind uses Valgrind's function wrapping mechanism to redirect calls to C math functions. Most of them map a single argument of type `double`, `float` or `long double` to a return value of the same type. The functions `atan2`/`atan2f`/`atan2l`, `fmod`/`fmodf`/`fmodl`, and `pow`/`powf`/`powl` take a second floating-point argument, `frexp`/`frexpf`/`frexpl` take a second argument of type `int*`, and `ldexp`/`ldexpf`/`ldexpl` take a second argument of type `int`.

Therefore, we have added the respective equivalents to `CALL_FN_W_WW` to `valgrind.h`. For example, Listing B.3 shows the definition of the macro `CALL_FN_D_D` to call the original function with a single argument of type `double`: This argument is stored in

Listing B.3: Definition of the macro `CALL_FN_D_D` for the x86-64 ISA, adapted from existing functions like `CALL_FN_W_WW` in Listing B.2 and added to Derivgrind's modified `valgrind.h`.

```
#define CALL_FN_FP_FP(lval, orig, outputreg, argvec_def, pushes) \
   do {                                                         \
      volatile OrigFn        _orig = (orig);                    \
      argvec_def                                   \
      _argvec[0] = (unsigned long)_orig.nraddr;  \
      __asm__ volatile(                                         \
         VALGRIND_CFI_PROLOGUE                                  \
         VALGRIND_ALIGN_STACK                                   \
         pushes \
         "movq (%%rax), %%rax\n\t"  /* target->%rax */          \
         VALGRIND_CALL_NOREDIR_RAX  \
         VALGRIND_RESTORE_STACK                                 \
         VALGRIND_CFI_EPILOGUE                                  \
         : /*out*/   outputreg (_res)                                \
         : /*in*/    "a" (&_argvec[0]) __FRAME_POINTER          \
         : /*trash*/ "cc", "memory", __CALLER_SAVED_REGS, "r14", "r15" \
      );      \
      lval = *(__typeof__(lval)*)& _res;                           \
   } while (0)

#define CALL_FN_D_D(lval, orig, arg1)                           \
  CALL_FN_FP_FP(lval, orig, "=Yz",  \
  volatile unsigned long _res; \
  volatile unsigned long _argvec[2];                      \
  _argvec[1] = *(unsigned long*)&(arg1);,                       \
  "subq $128,%%rsp\n\t movsd 8(%%rax), %%xmm0\n\t"  )
```

`_argvec[1]` in the C code, and subsequently accessible as `8(%rax)` in the assembly code, from where it is copied to XMM0 with a `movsd` instruction and made available to the C code via the output operand specification `"=Yz" (_res)`.

In order to be on the safe side, we have added the registers XMM1 to XMM15 to `__CALLER_SAVED_REGS`.

Similar work has also been performed for the x86 function wrapping macros.

B.2.c. Math Function Wrapper Example. Listing 7.1 in Chapter 7 shows the full code of Derivgrind's function wrapper for the `sin` function in `libm.so*`.

B.3. LibreOffice Calc Macros for Client Requests

After placing the Python file in Listing B.4 in a LibreOffice Calc macro search directory, four macros corresponding to the four Python functions of the same name can be selected in Calc's *Run Macro...* dialog (Figure 8.1). The four functions use `_cellMarkedByUser`, adapted from an online tutorial[77], to learn about the sheet, row and column of the table cell that the user selected prior to running the macro. This information allows to access the value of the table cell, and use the Python client request wrappers from Table 8.1.

Listing B.4: LibreOffice Calc macros, written in Python, to perform client requests on behalf of LibreOffice Calc. The code of `_cellMarkedByUser` has been adapted from an online tutorial[77].

```
import derivgrind
from com.sun.star.awt import MessageBoxButtons as MSG_BUTTONS

def _cellMarkedByUser():
    desktop = XSCRIPTCONTEXT.getDesktop()
    doc = XSCRIPTCONTEXT.getDocument()
    model = desktop.getCurrentComponent()
    sheet = model.CurrentController.ActiveSheet
    selection = doc.CurrentController.getSelection()
    row = selection.getRangeAddress().StartRow
    col = selection.getRangeAddress().StartColumn
    return sheet, row, col

def _warnNoFloatingPointContent():
    toolkit = XSCRIPTCONTEXT.getComponentContext().ServiceManager.createInstance↩
        ('com.sun.star.awt.Toolkit')
    parent = toolkit.getDesktopWindow()
    mb = toolkit.createMessageBox(parent, 'infobox', MSG_BUTTONS.BUTTONS_OK, "↩
        Error", "No floating-point content in cell ("+str(col+1)+","+str(row)+")↩
        .")
    return mb.execute()

def SetDotValue():
    sheet, row, col = _cellMarkedByUser()
    cellDot = sheet.getCellByPosition(col+1,row)
    if cellDot.Type.value in ["FORMULA", "VALUE"]:
        dot = cellDot.Value
        cellVal = sheet.getCellByPosition(col,row)
        cellVal.Value = derivgrind.set_dotvalue(cellVal.Value, dot)
    else:
        _warnNoFloatingPointContent()
    return None

def GetDotValue():
    sheet, row, col = _cellMarkedByUser()
    cellVal = sheet.getCellByPosition(col,row)
    if cellVal.Type.value in ["FORMULA", "VALUE"]:
        val = cellVal.Value
        cellDot = sheet.getCellByPosition(col+1,row)
        cellDot.Value = derivgrind.get_dotvalue(cellVal.Value)
    else:
        _warnNoFloatingPointContent()
    return None

def InputF():
    sheet, row, col = _cellMarkedByUser()
    cellVal = sheet.getCellByPosition(col,row)
    if cellVal.Type.value in ["FORMULA", "VALUE"]:
        val = cellVal.Value
        val = derivgrind.inputf(val)
        cellVal.Value = val
    else:
        _warnNoFloatingPointContent()
    return None

def OutputF():
    sheet, row, col = _cellMarkedByUser()
    cellVal = sheet.getCellByPosition(col,row)
    if cellVal.Type.value in ["FORMULA", "VALUE"]:
        val = cellVal.Value
        derivgrind.outputf(val)
    else:
        _warnNoFloatingPointContent()
    return None

g_exportedScripts = SetDotValue,GetDotValue,InputF,OutputF
```

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] Yves Achdou and Olivier Pironneau. *Computational Methods for Option Pricing*. Society for Industrial and Applied Mathematics, 2005. ISBN 9780898715736 9780898717495. doi: 10.1137/1.9780898717495. URL `http://epubs.siam.org/doi/book/10.1137/1.9780898717495`.

[3] Max Aehle, Johan Alme, Gergely Gábor Barnaföldi, Johannes Blühdorn, Tea Bodova, Vyacheslav Borshchov, Anthony van den Brink, Mamdouh Chaar, Viljar Eikeland, Gregory Feofilov, Christoph Garth, Nicolas R. Gauger, Georgi Genov, Ola Grøttvik, Håvard Helstrup, Sergey Igolkin, Ralf Keidel, Chinorat Kobdaj, Tobias Kortus, Viktor Leonhardt, Shruti Mehendale, Raju Ningappa Mulawade, Odd Harald Odland, George O'Neill, Gábor Papp, Thomas Peitzmann, Helge Egil Seime Pettersen, Pierluigi Piersimoni, Rohit Pochampalli, Maksym Protsenko, Max Rauch, Attiq Ur Rehman, Matthias Richter, Dieter Röhrich, Max Sagebaum, Joshua Santana, Alexander Schilling, Joao Seco, Arnon Songmoolnak, Jarle Rambo Sølie, Ganesh Tambave, Ihor Tymchuk, Kjetil Ullaland, Mónika Varga-Kőfaragó, Lennart Volz, Boris Wagner, Steffen Wendzel, Alexander Wiebel, RenZheng Xiao, Shiming Yang, Hiroki Yokoyama, and Sebastian Zillien. Derivatives in Proton CT. arXiv: 2202.05551, 2022. URL `https://arxiv.org/pdf/2202.05551.pdf`.

[4] Max Aehle, Johannes Blühdorn, Max Sagebaum, and Nicolas R. Gauger. Forward-Mode Automatic Differentiation of Compiled Programs, September 2022. URL `http://arxiv.org/abs/2209.01895`. arXiv:2209.01895 [cs].

[5] Max Aehle, Johannes Blühdorn, Max Sagebaum, and Nicolas R. Gauger. Reverse-Mode Automatic Differentiation of Compiled Programs, December 2022. URL `https://arxiv.org/pdf/2212.13760.pdf`. arXiv:2212.13760 [cs].

[6] Max Aehle, Johan Alme, Gergely Gábor Barnaföldi, Johannes Blühdorn, Tea Bodova, Vyacheslav Borshchov, Anthony van den Brink, Viljar Eikeland, Gregory Feofilov, Christoph Garth, Nicolas R Gauger, Ola Grøttvik, Håvard Helstrup, Sergey Igolkin, Ralf Keidel, Chinorat Kobdaj, Tobias Kortus, Lisa Kusch, Viktor Leonhardt, Shruti Mehendale, Raju Ningappa Mulawade, Odd Harald Odland, George O'Neill, Gábor Papp, Thomas Peitzmann, Helge Egil Seime Pettersen, Pierluigi Piersimoni, Rohit Pochampalli, Maksym Protsenko, Max Rauch, Attiq Ur Rehman, Matthias Richter, Dieter Röhrich, Max Sagebaum, Joshua Santana, Alexander Schilling, Joao Seco, Arnon Songmoolnak, Ákos Sudár, Ganesh Tambave, Ihor Tymchuk, Kjetil Ullaland, Mónika Varga-Kőfaragó, Lennart Volz, Boris Wagner, Steffen Wendzel, Alexander Wiebel, RenZheng Xiao, Shiming Yang, and Sebastian Zillien. Exploration of differentiability in a proton computed tomography simulation framework. *Physics in Medicine & Biology*, 68 (24):244002, December 2023. doi: 10.1088/1361-6560/ad0bdd. URL `https://dx.doi.org/10.1088/1361-6560/ad0bdd`.

[7] Max Aehle, Lorenzo Arsini, R. Belén Barreiro, Anastasios Belias, Florian Bury, Susana Cebrian, Alexander Demin, Jennet Dickinson, Julien Donini, Tommaso Dorigo, Michele Doro, Nicolas R. Gauger, Andrea Giammanco, Lindsey Gray, Borja S. González, Verena Kain, Jan Kieseler, Lisa Kusch, Marcus Liwicki, Gernot Maier, Federico Nardi, Fedor Ratnikov, Ryan Roussel, Roberto Ruiz de Austri, Fredrik Sandin, Michael Schenk, Bruno Scarpa, Pedro Silva, Giles C. Strong, and Pietro Vischia. Progress in End-to-End Optimization of Detectors for Fundamental Physics with Differentiable Programming, 2023. arXiv:2310.05673 [physics.ins-det].

[8] Max Aehle, Mihály Novák, Vassil Vassilev, Nicolas R. Gauger, Lukas Heinrich, Michael Kagan, and David Lange. Optimization Using Pathwise Algorithmic Derivatives of Electromagnetic Shower Simulations, 2024. arXiv:2405.07944 [physics.comp-ph].

[9] Gianluca Aglieri Rinella. The ALPIDE pixel sensor chip for the upgrade of the ALICE Inner Tracking System. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 845: 583–587, February 2017. ISSN 01689002. doi: 10.1016/j.nima.2016.05.016. URL `https://linkinghub.elsevier.com/retrieve/pii/S0168900216303825`.

[10] Stefano Agostinelli et al. GEANT4 — a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506:250, 2003. doi: 10.1016/S0168-9002(03)01368-8.

[11] Tim Albring, Max Sagebaum, and Nicolas R. Gauger. Efficient Aerodynamic Design using the Discrete Adjoint Method in SU2. *AIAA 2016-3518*, 2016.

[12] John Allison et al. Geant4 developments and applications. *IEEE Transactions on Nuclear Science*, 53(1):270–278, 2006. ISSN 0018-9499. doi: 10.1109/TNS.2006.869826. URL `http://ieeexplore.ieee.org/document/1610988/`.

[13] John Allison et al. Recent developments in Geant4. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 835:186–225, 2016. ISSN 01689002. doi: 10.1016/j.nima.2016.06.125. URL `https://linkinghub.elsevier.com/retrieve/pii/S0168900216306957`.

[14] Johan Alme, Gergely Gábor Barnaföldi, Rene Barthel, Vyacheslav Borshchov, Tea Bodova, Anthony van den Brink, Stephan Brons, Mamdouh Chaar, Viljar Eikeland, Grigory Feofilov, Georgi Genov, Silje Grimstad, Ola Grøttvik, Håvard Helstrup, Alf Herland, Annar Eivindplass Hilde, Sergey Igolkin, Ralf Keidel, Chinorat Kobdaj, Naomi van der Kolk, Oleksandr Listratenko, Qasim Waheed Malik, Shruti Mehendale, Ilker Meric, Simon Voigt Nesbø, Odd Harald Odland, Gábor Papp, Thomas Peitzmann, Helge Egil Seime Pettersen, Pierluigi Piersimoni, Maksym Protsenko, Attiq Ur Rehman, Matthias Richter, Dieter Röhrich, Andreas Tefre Samnøy, Joao Seco, Lena Setterdahl, Hesam Shafiee, Øistein Jelmert Skjolddal, Emilie Solheim, Arnon Songmoolnak, Ákos Sudár, Jarle Rambo Sølie, Ganesh Tambave, Ihor Tymchuk, Kjetil Ullaland, Håkon Andreas Underdal, Mónika Varga-Kőfaragó, Lennart Volz, Boris Wagner, Fredrik Mekki Widerøe, RenZheng Xiao, Shiming Yang, and Hiroki Yokoyama. A High-Granularity Digital Tracking Calorimeter Optimized for Proton CT. *Frontiers in Physics*, 8:460, 2020. ISSN 2296-424X. doi: 10.3389/fphy.2020.568243. URL `https://www.frontiersin.org/article/10.3389/fphy.2020.568243`.

[15] AMD64 Technology. AMD64 Architecture Programmer's Manual, Volumes 1–5. Technical report, October 2021. Revision 4.04.

[16] John O. Archambeau, Jerry D. Slater, James M. Slater, and Rosemary Tangeman. Role for proton beam irradiation in treatment of pediatric cns malignancies. *International Journal of Radiation Oncology*Biology*Physics*, 22(2):287–294, 1992. ISSN 0360-3016. doi: https://doi.org/10.1016/0360-3016(92)90045-J. URL `https://www.sciencedirect.com/science/article/pii/036030169290045J`.

[17] Gaurav Arya, Moritz Schauer, Frank Schäfer, and Christopher Rackauckas. Automatic Differentiation of Programs with Discrete Randomness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 10435–10447. Curran Associates, Inc., 2022. URL `https://proceedings.neurips.cc/paper_files/paper/2022/file/43d8e5fc816c692f342493331d5e98fc-Paper-Conference.pdf`.

[18] Matthias Baeten, Bert Mortier, Tine Baelmans, and Giovanni Samaey. Improved correlated sampling for iteratively coupled pde/monte-carlo methods used in plasma edge simulations. In *International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering, Date: 2017/04/16-2017/04/20, Location: Jeju, Korea*, 2017.

[19] Ahmet Refik Bahadır. A fully implicit finite-difference scheme for two-dimensional Burgers' equations. *Applied Mathematics and Computation*, 137(1):131–137, 2003. ISSN 0096-3003. doi: https://doi.org/10.1016/S0096-3003(02)00091-7. URL `https://www.sciencedirect.com/science/article/pii/S0096300302000917`.

[20] Brian C. Baumann, Nandita Mitra, Joanna G. Harton, Ying Xiao, Andrzej P. Wojcieszynski, Peter E. Gabriel, Haoyu Zhong, Huaizhi Geng, Abigail Doucette, Jenny Wei, Peter J. O'Dwyer, Justin E. Bekelman, and James M. Metz. Comparative Effectiveness of Proton vs Photon Therapy as Part of Concurrent Chemoradiotherapy for Locally Advanced Cancer. *JAMA Oncology*, 6(2):237–246, February 2020. ISSN 2374-2437. doi: 10.1001/jamaoncol.2019.4889. URL `https://doi.org/10.1001/jamaoncol.2019.4889`.

[21] Thomas M. Baumann and José Gracia. Cudagrind: Memory-Usage Checking for CUDA. In Andreas Knüpfer, José Gracia, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2013*, pages 67–78, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08144-1.

[22] Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18(1):5595–5637, 2017. ISSN 1532-4435.

[23] Atılım Güneş Baydin, Kyle Cranmer, Pablo de Castro Manzano, Christophe Delaere, Denis Derkach, Julien Donini, Tommaso Dorigo, Andrea Giammanco, Jan Kieseler, Lukas Layer, Gilles Louppe, Fedor Ratnikov, Giles C. Strong, Mia Tosi, Andrey Ustyuzhanin, Pietro Vischia, and Hevjin Yarar. Toward Machine Learning Optimization of Experimental Design. *Nuclear Physics News*, 31(1):25–28, 2021. doi: 10.1080/10619127.2021.1881364. URL `https://doi.org/10.1080/10619127.2021.1881364`.

[24] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A Dynamic Program Analysis to Find Floating-Point Accuracy Problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 453–462, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312059. doi: 10.1145/2254064.2254118. URL `https://doi.org/10.1145/2254064.2254118`.

[25] Berne convention for the Protection of Literary and Artistic Works. Convention of 1886, last amended on September 28, 1979.

[26] Jafar Biazar and Hossein Aminikhah. Exact and numerical solutions for non-linear Burger's equation by VIM. *Mathematical and Computer Modelling*, 49(7):1394–1400, 2009. ISSN 0895-7177. doi: https://doi.org/10.1016/j.mcm.2008.12.006. URL `https://www.sciencedirect.com/science/article/pii/S0895717709000107`.

[27] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR—Generating Derivative Codes from Fortran Programs. *Scientific Programming*, 1(1):11–29, January 1992. doi: 10.1155/1992/717832. URL `https://doi.org/10.1155/1992/717832`.

[28] Christian Bischof, George Corliss, Lawrence Green, Andreas Griewank, Kara Haigler, and Perry Newman. Automatic differentiation of advanced CFD codes for multidisciplinary design. *Computing Systems in Engineering*, 3(6):625–637, 1992. ISSN 0956-0521. doi: https://doi.org/10.1016/0956-0521(92)90014-A. URL `https://www.sciencedirect.com/science/article/pii/095605219290014A`.

[29] Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. Adifor 2.0: automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3):18–32, 1996. doi: 10.1109/99.537089.

[30] Christian Bischof, Niels Guertler, Andreas Kowarz, and Andrea Walther. Parallel Reverse Mode Automatic Differentiation for OpenMP Programs with ADOL-C. In Christian H. Bischof, H. Martin Bücker, Paul Hovland, Uwe Naumann, and Jean Utke, editors, *Advances in Automatic Differentiation*, pages 163–173, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-68942-3.

[31] Christian H. Bischof, H. Martin Bücker, Arno Rasch, Emil Slusanschi, and Bruno Lang. Automatic Differentiation of the General-Purpose Computational Fluid Dynamics Package FLUENT. *Journal of Fluids Engineering*, 129(5):652–658, 2007. ISSN 0098-2202, 1528-901X. doi: 10.1115/1.2720475. URL `https://asmedigitalcollection.asme.org/fluidsengineering/article/129/5/652/443979/Automatic-Differentiation-of-the-GeneralPurpose`.

[32] Johannes Blühdorn, Max Sagebaum, and Nicolas R. Gauger. Event-Based Automatic Differentiation of OpenMP with OpDiLib. *ACM Transactions on Mathematical Software*, 49(1), March 2023. ISSN 0098-3500. doi: 10.1145/3570159. URL `https://doi.org/10.1145/3570159`.

[33] Rocco Bombardieri, Rauno Cavallaro, Ruben Sanchez, and Nicolas R. Gauger. Aerostructural wing shape optimization assisted by algorithmic differentiation. *Structural and Multidisciplinary Optimization*, 64(2):739–760, 2021. ISSN 1615-147X, 1615-1488. doi: 10.1007/s00158-021-02884-5. URL `https://link.springer.com/10.1007/s00158-021-02884-5`.

[34] Thomas Bortfeld. An analytical approximation of the bragg curve for therapeutic proton beams. *Medical Physics*, 24(12):2024–2033, 1997. ISSN 00942405. doi: 10.1118/1.598116. URL `http://doi.wiley.com/10.1118/1.598116`.

[35] Joël Brezillon and Richard P. Dwight. Applications of a discrete viscous adjoint method for aerodynamic shape optimisation of 3D configurations. *CEAS Aeronautical Journal*, 3(1):25–34, 2012. ISSN 1869-5582, 1869-5590. doi: 10.1007/s13272-011-0038-0. URL `http://link.springer.com/10.1007/s13272-011-0038-0`.

[36] Derek Bruening and Qin Zhao. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, page 213–223, USA, 2011. IEEE Computer Society. ISBN 9781612843568.

[37] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[38] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *International Symposium on Code Generation and Optimization (CGO-03)*, March 2003.

[39] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, page 133–144, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311762. doi: 10.1145/2151024.2151043. URL `https://doi.org/10.1145/2151024.2151043`.

[40] H. Martin Bücker, Arno Rasch, and Andreas Wolf. A class of OpenMP applications involving nested parallelism. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, page 220–224, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138121. doi: 10.1145/967900.967948. URL `https://doi.org/10.1145/967900.967948`.

[41] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*, pages 749–754, 2006. doi: 10.1109/ISCC.2006.158.

[42] Bruce Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3(4):311–326, 1994. ISSN 1055-6788, 1029-4937. doi: 10.1080/10556789408805572. URL `http://www.tandfonline.com/doi/abs/10.1080/10556789408805572`.

[43] Allan MacLeod Cormack. Representation of a Function by Its Line Integrals, with Some Radiological Applications. *Journal of Applied Physics*, 34(9):2722–2727, 1963. doi: 10.1063/1.1729798. URL `https://doi.org/10.1063/1.1729798`.

[44] Michael F. Cowlishaw. Decimal floating-point: algorism for computers. In *Proceedings 2003 16th IEEE Symposium on Computer Arithmetic*, pages 104–111, 2003. doi: 10.1109/ARITH.2003.1207666.

[45] Karl Cronburg. GitHub repository `shadow-memory`, 2022. `https://github.com/cronburg/shadow-memory`.

[46] Benjamin Dauvergne and Laurent Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994, pages 566–573. Springer Berlin Heidelberg, 2006. doi: 10.1007/11758549_78.

[47] Bruce Dawson. Intel Underestimates Error Bounds by 1.3 quintillion, October 2014. URL `https://randomascii.wordpress.com/2014/10/09/intel-underestimates-error-bounds-by-1-3-quintillion/`.

[48] George Dedes, Jannis Dickmann, Katharina Niepel, Philipp Wesp, Robert P Johnson, Mark Pankuch, Vladimir Bashkirov, Simon Rit, Lennart Volz, Reinhard W Schulte, Guillaume Landry, and Katia Parodi. Experimental comparison of proton CT and dual energy x-ray CT for relative stopping power estimation in proton therapy. *Physics in Medicine & Biology*, 64(16):165002, 2019. ISSN 1361-6560. doi: 10.1088/1361-6560/ab2b72. URL `https://iopscience.iop.org/article/10.1088/1361-6560/ab2b72`.

[49] Ethan A. DeJongh, Don F. DeJongh, Igor Polnyi, Victor Rykalin, Christina Sarosiek, George Coutrakon, Kirk L. Duffin, Nicholas T. Karonis, Caesar E. Ordoñez, Mark Pankuch, John R. Winans, and James S. Welsh. Technical Note: A fast and monolithic prototype clinical proton radiography system optimized for pencil beam scanning. *Medical Physics*, 48(3):1356–1364, March 2021. ISSN 0094-2405, 2473-4209. doi: 10.1002/mp.14700. URL `https://onlinelibrary.wiley.com/doi/10.1002/mp.14700`.

[50] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, RFC Editor, May 1996. URL `https://www.rfc-editor.org/rfc/rfc1951.txt`.

[51] Tommaso Dorigo, Andrea Giammanco, Pietro Vischia, Max Aehle, Mateusz Bawaj, Alexey Boldyrev, Pablo de Castro Manzano, Denis Derkach, Julien Donini, Auralee Edelen, Federica Fanzago, Nicolas R. Gauger, Christian Glaser, Atılım G. Baydin, Lukas Heinrich, Ralf Keidel, Jan Kieseler, Claudius Krause, Maxime Lagrange, Max Lamparth, Lukas Layer, Gernot Maier, Federico Nardi, Helge E.S. Pettersen, Alberto Ramos, Fedor Ratnikov, Dieter Röhrich, Roberto Ruiz de Austri, Pablo Martínez Ruiz del Árbol, Oleg Savchenko, Nathan Simpson, Giles C. Strong, Angela Taliercio, Mia Tosi, Andrey Ustyuzhanin, and Haitham Zaraket. Toward the end-to-end optimization of particle physics instruments with differentiable programming. *Reviews in Physics*, 10:100085, 2023. ISSN 2405-4283. doi: https://doi.org/10.1016/j.revip.2023.100085. URL `https://www.sciencedirect.com/science/article/pii/S2405428323000047`.

[52] Will Drewry and Tavis Ormandy. Flayer: Exposing Application Internals. In *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT '07)*, August 2007.

[53] A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic. Decimal floating-point in z9: An implementation and testing perspective. *IBM Journal of Research and Development*, 51(1.2):217–227, 2007. doi: 10.1147/rd.511.0217.

[54] EPO Board of Appeal, consisting of P. K. J. van den Berg (Chairman), V. Di Cerbo, and R. R. K. Zimmermann. Decision of Technical Board of Appeal 3.5.1 dated 1 July 1998 - T 1173/97 - 3.5.1. *Official Journal of the European Patent Office*, pages 609–632, October 1999.

[55] Michela Esposito, Chris Waltham, Jonathan T. Taylor, Sam Manger, Ben Phoenix, Tony Price, Gavin Poludniowski, Stuart Green, Philip M. Evans, Philip P. Allport, Spyros Manolopulos, Jaime Nieto-Camero, Julyan Symons, and Nigel M. Allinson. PRaVDA: The first solid-state system for proton computed tomography. *Physica Medica*, 55:149–154, 2018. ISSN 11201797. doi: 10.1016/j.ejmp.2018.10.020. URL `https://linkinghub.elsevier.com/retrieve/pii/S1120179718313073`.

[56] Steve Fanning, Vasudev Narayanan, Olivier Hallot, Jean H. Weber, et al. LibreOffice 7.3 Base Guide. August 2022.

[57] Warren Ferguson, Marius Cornea, Cristina Anderson, and Eric Schneider. The Difference Between x87 Instructions FSIN, FCOS, FSINCOS, and FPTAN and Mathematical Functions sin, cos, sincos, and tan, 2015. URL `https://software.intel.com/content/dam/develop/external/us/en/documents/x87trigonometricinstructionsvsmathfunctions.pdf`.

[58] François Févotte and Bruno Lathuilière. Studying the Numerical Quality of an Industrial Computing Code: A Case Study on code_aster. In *10th International Workshop on Numerical Software Verification (NSV)*, pages 61–80, Heidelberg, Germany, July 2017. ISBN 978-3-319-63501-9. doi: 10.1007/978-3-319-63501-9_5.

[59] Ailton Santos Filho, Ricardo J. Rodríguez, and Eduardo L. Feitosa. Evasion and Countermeasures Techniques to Detect Dynamic Binary Instrumentation Frameworks. *Digital Threats*, 3(2), February 2022. ISSN 2692-1626. doi: 10.1145/3480463. URL `https://doi.org/10.1145/3480463`.

[60] Herbert Fischer. Automatic differentiation of the vector that solves a parametric linear system. *Journal of Computational and Applied Mathematics*, 35(1):169–184, 1991. ISSN 0377-0427. doi: https://doi.org/10.1016/0377-0427(91)90205-X. URL `https://www.sciencedirect.com/science/article/pii/037704279190205X`.

[61] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, 33(2):13–es, June 2007. ISSN 0098-3500. doi: 10.1145/1236463.1236468. URL `https://doi.org/10.1145/1236463.1236468`.

[62] Free Software Foundation, Inc. GNU General Public License – Version 2, June 1991, . URL `https://www.gnu.org/licenses/old-licenses/gpl-2.0.html`.

[63] Free Software Foundation, Inc. GNU General Public License – Version 3, 29 June 2007, . URL `https://www.gnu.org/licenses/gpl-3.0.html`.

[64] Free Software Foundation, Inc. GNU Lesser General Public License – Version 2.1, February 1999, . URL `https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`.

[65] Free Software Foundation, Inc. GNU Lesser General Public License – Version 3, 29 June 2007, . URL `https://www.gnu.org/licenses/lgpl-3.0.html`.

[66] Fujitsu Limited. SPARC64(TM) X / X+ Specification, January 2015. Version 29.0.

[67] François Févotte and Bruno Lathuilière. VERROU: a CESTAC evaluation without recompilation. In *International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)*, Uppsala, Sweden, September 2016.

[68] François Févotte and Bruno Lathuilière. Debugging and Optimization of HPC Programs with the Verrou Tool. In *International Workshop on Software Correctness for HPC Applications (Correctness)*, Denver, CO, USA, November 2019. doi: 10.1109/Correctness49594.2019.00006.

[69] Dmitrij Gendler, Uwe Naumann, and Bruce Christianson. Automatic Differentiation of Assembler Code. In *Proceedings of the IADIS International Conference on Applied Computing*, pages 431–436. IADIS, 2007.

[70] Jean Charles Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1(1):13–21, 1992. doi: 10.1080/10556789208805503.

[71] Peter W. Glynn. Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM*, 33(10):75–84, October 1990. ISSN 0001-0782. doi: 10.1145/84537.84552. URL `https://doi.org/10.1145/84537.84552`.

[72] Godbolt query. URL `https://godbolt.org/z/ozTz5Gvce`.

[73] Dmitri Goloubentsev and Evgeny Lakshtanov. AAD: Breaking the Primal Barrier. *Wilmott*, 2019(103):8–11, 2019. doi: https://doi.org/10.1002/wilm.10785. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/wilm.10785`.

[74] Hadrien Grasland, François Févotte, Bruno Lathuilière, and David Chamont. Floating-point profiling of ACTS using Verrou. *EPJ Web of Conferences*, 214, 2019. doi: 10.1051/epjconf/201921405025.

[75] Andreas Griewank and Andrea Walther. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, second edition, 2008. doi: 10.1137/1.9780898717761. URL `https://epubs.siam.org/doi/abs/10.1137/1.9780898717761`.

[76] Pin-Wen Guan. Differentiable thermodynamic modeling. *Scripta Materialia*, 207: 114217, 2022. ISSN 1359-6462. doi: https://doi.org/10.1016/j.scriptamat.2021.11

4217. URL https://www.sciencedirect.com/science/article/pii/S1359646221004978.

[77] Gwenole Capp. LibreOffice Calc & Python Programming: part 6 – Type of a Cell content, June 2020. URL https://tutolibro.tech/2020/06/26/libreoffice-calc-python-programming-part-6-type-of-a-cell-content/.

[78] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[79] Laurent Hascoët and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):1–43, 2013. ISSN 0098-3500, 1557-7295. doi: 10.1145/2450153.2450158. URL https://dl.acm.org/doi/10.1145/2450153.2450158.

[80] Marcel Heing-Becker, Timo Kamph, and Sibylle Schupp. Bit-error injection for software developers. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 434–439, 2014. doi: 10.1109/CSMR-WCRE.2014.6747212.

[81] Marc Henrard. Calibration in Finance: Very Fast Greeks Through Algorithmic Differentiation and Implicit Function. *Procedia Computer Science*, 18:1145–1154, 2013. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2013.05.280. URL https://www.sciencedirect.com/science/article/pii/S1877050913004237. 2013 International Conference on Computational Science.

[82] Jack P. Higgins, Michael B. Bernstein, and James W. Hodge. Enhancing immune responses to tumor-associated antigens. *Cancer Biology & Therapy*, 8(15):1440–1449, 2009. doi: 10.4161/cbt.8.15.9133. PMID: 19556848.

[83] Godfrey Newbold Hounsfield. Computerized transverse axial scanning (tomography): Part I. Description of system. *British Journal of Radiology*, 68, 1973.

[84] Jan Hückelheim and Laurent Hascoët. Source-to-Source Automatic Differentiation of OpenMP Parallel Loops. *ACM Transactions on Mathematical Software*, 48(1), February 2022. ISSN 0098-3500. doi: 10.1145/3472796. URL https://doi.org/10.1145/3472796.

[85] Jan Hückelheim, Harshitha Menon, William Moses, Bruce Christianson, Paul Hovland, and Laurent Hascoët. Understanding Automatic Differentiation Pitfalls, 2023. arXiv:2305.07546 [math.NA].

[86] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi: 10.1109/JRPROC.1952.273898.

[87] Ford Hurley, Reinhard Wilhelm Schulte, Vladimir A. Bashkirov, Andrew J. Wroe, Abiel Ghebremedhin, Hartmut F.-W. Sadrozinski, Victor Rykalin, George Coutrakon, Pete Koss, and Baldev Patyal. Water-equivalent path length calibration of a prototype proton CT scanner: Water-equivalent path length calibration for proton CT. *Medical Physics*, 39(5):2438–2446, 2012. ISSN 00942405. doi: 10.1118/1.3700173. URL `http://doi.wiley.com/10.1118/1.3700173`.

[88] Alexander Hück, Christian Bischof, and Jean Utke. Checking C++ codes for compatibility with operator overloading. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 91–100, 2015. doi: 10.1109/SCAM.2015.7335405.

[89] IBM z9 EC and z9 BC – Delivering greater value for everyone. IBM United States Announcement 107-190, April 18, 2007.

[90] IEEE 754. IEEE Standard for Floating-Point Arithmetic. Technical report, 2008. URL `http://ieeexplore.ieee.org/document/4610935/`.

[91] Michael Innes. Don't Unroll Adjoint: Differentiating SSA-Form Programs. Technical Report arXiv:1810.07951, arXiv, March 2019. URL `http://arxiv.org/abs/1810.07951`.

[92] ISO. *ISO/IEC 9899:TC3*. International Organization for Standardization, Geneva, Switzerland, September 2007. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`. Document number N1256, draft for C 99 with Technical Corrigendum 3.

[93] ISO. *Programming languages – C*. International Organization for Standardization, Geneva, Switzerland, April 2011. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`. Document number N1570, draft for C 11.

[94] ISO. *Working Draft, Standard for Programming Language C++*. International Organization for Standardization, Geneva, Switzerland, February 2012. URL `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`. Document number N3242=11-0012, draft for C++11.

[95] ISO. *Working Draft, Standard for Programming Language C++*. International Organization for Standardization, Geneva, Switzerland, October 2013. URL `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf`. Document number N3797, draft for C++14.

[96] ISO. *Working Draft, Standard for Programming Language C++*. International Organization for Standardization, Geneva, Switzerland, March 2017. URL `http`

s://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf. Document number N4659, draft for C++17.

[97] ISO. *Working Draft, Standard for Programming Language C++*. International Organization for Standardization, Geneva, Switzerland, January 2020. URL https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4849.pdf. Document number N4849, draft for C++20.

[98] ISO. *ISO/IEC 9899:2023 (E)*. International Organization for Standardization, Geneva, Switzerland, April 2023. URL https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf. Document number N3096, draft for C 23.

[99] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – Seamless operability between C++11 and Python, 2017. https://github.com/pybind/pybind11.

[100] S. Jan et al. GATE - Geant4 Application for Tomographic Emission: a simulation toolkit for PET and SPECT. *Physics in Medicine and Biology*, 49(19):4543–4561, 2004. ISSN 0031-9155. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3267383/.

[101] Louis Jenkins and Michael L. Scott. Persistent Memory Analysis Tool (PMAT), 2020.

[102] Robert P Johnson. Review of medical radiography and tomography with proton beams. *Reports on Progress in Physics*, 81(1):016701, 2018. ISSN 0034-4885, 1361-6633. doi: 10.1088/1361-6633/aa8b1d. URL https://iopscience.iop.org/article/10.1088/1361-6633/aa8b1d.

[103] Mateusz Jurczyk. Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking, June 2018.

[104] Mateusz Jurczyk and Gynvael Coldwind. Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns, April 2013.

[105] Michael Kagan and Lukas Heinrich. Branches of a tree: Taking derivatives of programs with discrete and branching randomness in high energy physics, 2023. arXiv:2308.16680 [stat.ML].

[106] Tomasz Kapela. An Introduction to pmemcheck (Part I) – Basics, July 2015. URL https://pmem.io/2015/07/17/pmemcheck-basic.html.

[107] David J. Kerr, Daniel G. Haller, Cornelis J. H. van de Velde, and Michael Baumann, editors. *Oxford Textbook of Oncology*. Oxford University Press, third edition, 2016.

[108] Wei Ming Khoo et al. Taintgrind: a Valgrind taint analysis tool. GitHub repository, https://github.com/wmkhoo/taintgrind.

[109] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. URL `http://arxiv.org/abs/1312.6114`.

[110] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi: 10.1109/CGO.2004.1281665.

[111] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. On correctness of automatic differentiation for non-differentiable functions. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

[112] Jonathan E Leeman, Paul B Romesser, Ying Zhou, Sean McBride, Nadeem Riaz, Eric Sherman, Marc A Cohen, Oren Cahlon, and Nancy Lee. Proton therapy for head and neck cancer: expanding the therapeutic window. *The Lancet Oncology*, 18(5):e254–e265, 2017. ISSN 1470-2045. doi: https://doi.org/10.1016/S1470-2045(17)30179-1. URL `https://www.sciencedirect.com/science/article/pii/S1470204517301791`.

[113] Randall J. LeVeque. *Numerical Methods for Conservation Laws.* Birkhäuser Basel, 1992. ISBN 9783764327231 9783034886291. doi: 10.1007/978-3-0348-8629-1. URL `http://link.springer.com/10.1007/978-3-0348-8629-1`.

[114] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable Monte Carlo ray tracing through edge sampling. *ACM Transactions on Graphics*, 37(6), dec 2018. ISSN 0730-0301. doi: 10.1145/3272127.3275109. URL `https://doi.org/10.1145/3272127.3275109`.

[115] LibreOffice Contributors. LibreOffice Core Source Code. `https://github.com/LibreOffice/core/`.

[116] Hui Liu and Joe Y Chang. Proton therapy in clinical practice. *Chinese Journal of Cancer*, 30(5):315, 2011.

[117] LLVM Contributors. The LLVM Compiler Infrastructure. URL `https://llvm.org/`. Webpage.

[118] H. J. Lu, David L Kreitzer, Milind Girkar, and Zia Ansari. System V Application Binary Interface, Intel386 Architecture Processor Supplement, Version 1.0. Technical report, February 2015. URL `http://www.uclibc.org/docs/psABI-i386.pdf`.

[119] Mathias Luers, Max Sagebaum, Sebastian Mann, Jan Backhaus, David Grossmann, and Nicolas R. Gauger. Adjoint-based volumetric shape optimization of turbine blades. In *2018 Multidisciplinary Analysis and Optimization Conference.* American

Institute of Aeronautics and Astronautics, 2018. ISBN 9781624105500. doi: 10.2514/6.2018-3638. URL `https://arc.aiaa.org/doi/10.2514/6.2018-3638`.

[120] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi: 10.1145/1065010.1065034. URL `https://doi.org/10.1145/1065010.1065034`.

[121] Iván Lux and László Koblinger. *Monte Carlo Particle Transport Methods: Neutron and Photon Calculations*. CRC Press, 1991. ISBN 0-8493-6074-9.

[122] Zhoujie Lyu, Gaetan K. Kenway, Cody Paige, and Joaquim R. R. A. Martins. Automatic Differentiation Adjoint of the Reynolds-Averaged Navier-Stokes Equations with a Turbulence Model. In *21st AIAA Computational Fluid Dynamics Conference*, 2013. doi: 10.2514/6.2013-2581. URL `https://arc.aiaa.org/doi/abs/10.2514/6.2013-2581`.

[123] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless Gradients in Numpy. In *ICML 2015 AutoML Workshop*, volume 238, page 5, 2015.

[124] Serena Mattiazzo, Filippo Baruffaldi, Dario Bisello, Benedetto Di Ruzza, Piero Giubilato, Roberto Iuppa, Chiara La Tessa, Devis Pantano, Nicola Pozzobon, Ester Ricci, Walter Snoeys, and Jeffery Wyss. iMPACT: An Innovative Tracker and Calorimeter for Proton Computed Tomography. *IEEE Transactions on Radiation and Plasma Medical Sciences*, 2(4):345–352, 2018. doi: 10.1109/TRPMS.2018.2825499.

[125] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft Version 0.99.6. Technical report, 2012. URL `https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf`.

[126] Sebastian Meyer, Jonathan Bortfeldt, Paulina Lämmer, Franz S Englbrecht, Marco Pinto, Katrin Schnürle, Matthias Würl, and Katia Parodi. Optimization and performance study of a proton CT system for pre-clinical small animal imaging. *Physics in Medicine & Biology*, 65(15):155008, 2020. ISSN 1361-6560. doi: 10.1088/1361-6560/ab8afc. URL `https://iopscience.iop.org/article/10.1088/1361-6560/ab8afc`.

[127] Mathieu Morlighem, Daniel Goldberg, Thiago Dias dos Santos, Jane Lee, and Max Sagebaum. Mapping the sensitivity of the Amundsen Sea Embayment to changes in external forcings using Automatic Differentiation. *Geophysical Research Letters*, 48(23), 2021. doi: https://doi.org/10.1029/2021GL095440.

[128] William Moses and Valentin Churavy. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12472–12485. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf`.

[129] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476165. URL `https://doi.org/10.1145/3458817.3476165`.

[130] William S. Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland. Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18, 2022. doi: 10.1109/SC41404.2022.00065.

[131] Jens-Dominik Müller and Paul Cusdin. On the performance of discrete adjoint CFD codes using automatic differentiation. *International Journal for Numerical Methods in Fluids*, 47(8):939–945, 2005. ISSN 0271-2091, 1097-0363. doi: 10.1002/fld.885. URL `https://onlinelibrary.wiley.com/doi/10.1002/fld.885`.

[132] Md. Naimuddin, George Coutrakon, Gerald Blazey, Swek Boi, Alexandre Dyshkant, Bela Erdelyi, David Hedin, Elizabeth Johnson, James Krider, V. Rukalin, Sergey A. Uzunyan, Vishnu Zutshi, R. Fordt, Greg Sellberg, J.E. Rauch, M. Roman, Paul Rubinov, and P. Wilson. Development of a proton Computed Tomography detector system. *Journal of Instrumentation*, 11(2):C02012–C02012, 2016. ISSN 1748-0221. doi: 10.1088/1748-0221/11/02/C02012. URL `https://iopscience.iop.org/article/10.1088/1748-0221/11/02/C02012`.

[133] Uwe Naumann and Jack Toit. Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance. *SSRN Scholarly Paper*, 21(4), 2018. URL `https://papers.ssrn.com/abstract=3122293`.

[134] Anil Nemili, Emre Özkaya, Nicolas R. Gauger, Felix Kramer, and Frank Thiele. Accurate discrete adjoint approach for optimal active separation control. *AIAA Journal*, 55(9):3016–3026, 2017. ISSN 0001-1452, 1533-385X. doi: 10.2514/1.J055009. URL `https://arc.aiaa.org/doi/10.2514/1.J055009`.

[135] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation, or Building Tools is Easy.* PhD thesis, Trinity College, University of Cambridge, 2004.

[136] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-Checking Entire Programs Without Recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, January 2004.

[137] Nicholas Nethercote and Alan Mycroft. Redux: A Dynamic Dataflow Tracer. *Electronic Notes in Theoretical Computer Science*, 89(2):149–170, 2003. ISSN 1571-0661. doi: https://doi.org/10.1016/S1571-0661(04)81047-8. URL https://www.sciencedirect.com/science/article/pii/S1571066104810478. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[138] Nicholas Nethercote and Julian Seward. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, page 65–74, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936301. doi: 10.1145/1254810.1254820. URL https://doi.org/10.1145/1254810.1254820.

[139] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250746. URL https://doi.org/10.1145/1273442.1250746.

[140] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2015.

[141] Mihály Novák. The HepEmShow R&D Project. URL https://github.com/mnovak42/hepemshow.

[142] Mihály Novák, Jonas Hahnfeld, et al. G4HepEm. URL https://github.com/mnovak42/g4hepem.

[143] OECD. *Health at a Glance 2021*. OECD Publishing, Paris, 2021. doi: https://doi.org/https://doi.org/10.1787/ae3016b9-en. URL https://www.oecd-ilibrary.org/content/publication/ae3016b9-en.

[144] Harald Paganetti. Range uncertainties in proton therapy and the role of Monte Carlo simulations. *Physics in Medicine and Biology*, 57(11):R99–R117, May 2012. doi: 10.1088/0031-9155/57/11/r99. URL https://doi.org/10.1088/0031-9155/57/11/r99.

[145] John Palmer. The Intel(R) 8087 Numeric Data Processor. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, ISCA '80, page 174–181, New York, NY, USA, 1980. Association for Computing Machinery. ISBN 9781450373906. doi: 10.1145/800053.801923. URL https://doi.org/10.1145/800053.801923.

[146] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[147] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. The Morgan Kaufmann series in computer architecture and design. Elsevier Inc., 5 edition, 2014.

[148] Peter Pemler, Jürgen Besserer, Jorrit de Boer, Matthias Dellert, C. Gahn, Martin Moosburger, Uwe Schneider, Eros Pedroni, and H. Stäuble. A detector system for proton radiography on the gantry of the paul-scherrer-institute. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 432(2):483–495, 1999. ISSN 0168-9002. doi: https://doi.org/10.1016/S0168-9002(99)00284-3. URL `https://www.sciencedirect.com/science/article/pii/S0168900299002843`.

[149] Scott Penfold and Yair Censor. Techniques in iterative proton CT image reconstruction. *Sensing and Imaging*, 16(1):19, 2015. ISSN 1557-2064, 1557-2072. doi: 10.1007/s11220-015-0122-3. URL `http://link.springer.com/10.1007/s11220-015-0122-3`.

[150] Helge Egil Seime Pettersen, Johan Alme, Aleksandra Biegun, Anthony van den Brink, Mamdouh Chaar, Dominik Fehlker, Ilker Meric, Odd Harald Odland, Thomas Peitzmann, Elena Rocco, Kjetil Ullaland, Hongkai Wang, Shiming Yang, Chunjian Zhang, and Dieter Röhrich. Proton tracking in a high-granularity Digital Tracking Calorimeter for proton CT purposes. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 860:51–61, 2017. ISSN 01689002. doi: 10.1016/j.nima.2017.02.007. URL `https://linkinghub.elsevier.com/retrieve/pii/S0168900217301882`.

[151] Helge Egil Seime Pettersen, Mamdouh Chaar, Ilker Meric, Odd Harald Odland, Jarle Rambo Sølie, and Dieter Röhrich. Accuracy of parameterized proton range models; a comparison. *Radiation Physics and Chemistry*, 144:295–297, March 2018. doi: 10.1016/j.radphyschem.2017.08.028. IF 1.984.

[152] Helge Egil Seime Pettersen, Johan Alme, Gergely Gábor Barnaföldi, Rene Barthel, Anthony van den Brink, Mamdouh Chaar, Viljar Eikeland, Alba García-Santos, Georgi Genov, Silje Grimstad, Ola Grøttvik, Håvard Helstrup, Kristin Fanebust Hetland, Shruti Mehendale, Ilker Meric, Odd Harald Odland, Gábor Papp, Thomas Peitzmann, Pierluigi Piersimoni, Attiq Ur Rehman, Matthias Richter,

Andreas Tefre Samnøy, Joao Seco, Hesam Shafiee, Eivind Vågslid Skjæveland, Jarle Rambo Sølie, Ganesh Tambave, Kjetil Ullaland, Mónika Varga-Kőfaragó, Lennart Volz, Boris Wagner, Shiming Yang, and Dieter Röhrich. Design optimization of a pixel-based range telescope for proton computed tomography. *Physica Medica*, 63:87–97, July 2019. ISSN 11201797. doi: 10.1016/j.ejmp.2019.05.026. URL https://linkinghub.elsevier.com/retrieve/pii/S1120179719301358.

[153] Helge Egil Seime Pettersen, Ilker Meric, Odd Harald Odland, Hesam Shafiee, Jarle Rambo Sølie, and Dieter Röhrich. Proton tracking algorithm in a pixel-based range telescope for proton computed tomography, 2020. URL http://arxiv.org/abs/2006.09751. arXiv:2006.09751 [physics].

[154] Danilo Piparo, Vincenzo Innocente, and Thomas Hauth. Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions. *Journal of Physics: Conference Series*, 513(5):052027, June 2014. ISSN 1742-6588, 1742-6596. doi: 10.1088/1742-6596/513/5/052027. URL https://iopscience.iop.org/article/10.1088/1742-6596/513/5/052027.

[155] Gavin Poludniowski, Nigel M Allinson, and Philip Evans. Proton radiography and tomography with application to proton therapy. *The British Journal of Radiology*, 88(1053):20150134, 2015. ISSN 0007-1285, 1748-880X. doi: 10.1259/bjr.20150134. URL http://www.birpublications.org/doi/10.1259/bjr.20150134.

[156] Max Sagebaum, Emre Özkaya, Nicolas R. Gauger, Jan Backhaus, Christian Frey, Sebastian Mann, and Marc Nagel. Efficient Algorithmic Differentiation Techniques for Turbo-machinery Design. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2017. doi: 10.2514/6.2017-3998. URL https://arc.aiaa.org/doi/abs/10.2514/6.2017-3998.

[157] Max Sagebaum, Tim Albring, and Nicolas R. Gauger. Expression templates for primal value taping in the reverse mode of algorithmic differentiation. *Optimization Methods and Software*, 33(4):1207–1231, 2018. ISSN 1055-6788, 1029-4937. doi: 10.1080/10556788.2018.1471140. URL https://www.tandfonline.com/doi/full/10.1080/10556788.2018.1471140.

[158] Max Sagebaum, Tim Albring, and Nicolas R. Gauger. High-Performance Derivative Computations Using CoDiPack. *ACM Transactions on Mathematical Software*, 45 (4), December 2019. ISSN 0098-3500. doi: 10.1145/3356900. URL https://doi.org/10.1145/3356900.

[159] Max Sagebaum, Johannes Blühdorn, and Nicolas R. Gauger. Index handling and assign optimization for Algorithmic Differentiation reuse index managers. arXiv cs.MS 2006.12992, 2021. URL https://arxiv.org/abs/2006.12992.

[160] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. Finding Root Causes of Floating Point Error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018,

page 256–269, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192411. URL `https://doi.org/10.1145/3192366.3192411`.

[161] Yuichi Saraya, Takuji Izumikawa, Jyun Goto, Takeo Kawasaki, and T. Kimura. Study of spatial resolution of proton computed tomography using a silicon strip detector. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 735:485–489, 2014. ISSN 01689002. doi: 10.1016/j.nima.2013.09.051. URL `https://linkinghub.elsevier.com/retrieve/pii/S0168900213012850`.

[162] Monica Scaringella, Mirko Brianzi, Mara Bruzzi, Marta Bucciolini, Massimo Carpinelli, Pablo G. A. Cirrone, Carlo Civinini, Giacomo Cuttone, Domenico Lo Presti, Stefania Pallotta, Cristina Pugliatti, Nunzio Randazzo, Francesco Romano, Valeria Sipala, Concetta Stancampiano, Cinzia Talamonti, Mauro Tesi, Eleonora Vanzi, and Margherita Zani. The PRIMA (PRoton IMAging) collaboration: Development of a proton Computed Tomography apparatus. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 730:178–183, 2013. ISSN 01689002. doi: 10.1016/j.nima.2013.05.181. URL `https://linkinghub.elsevier.com/retrieve/pii/S0168900213008036`.

[163] Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea. AD for an Array Language with Nested Parallelism, 2022. arXiv:2202.10297v1 [cs.PL].

[164] Johannes Schoder. Embedding automatic differentiation in a RISC-V processor design, 2023. Talk at the 26th EuroAD Workshop, 4th–5th December 2023, Aachen, Germany.

[165] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 3528–3536, Cambridge, MA, USA, 2015. MIT Press.

[166] Eric M Schwarz and Steven R Carlough. Power6 decimal divide. In *2007 IEEE International Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, pages 128–133. IEEE, 2007.

[167] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 2, USA, 2005. USENIX Association.

[168] Julian Seward, Nicholas Nethercote, Tom Hughes, Jeremy Fitzhardinge, Josef Weidendorfer, et al. Valgrind Documentation, Release 3.19.0, 11 Apr 2022, 2022. URL `https://sourceware.org/pub/valgrind/valgrind-3.19.0.tar.bz2`. We use this as a reference to Valgrind's source code as well.

[169] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372268. doi: 10.1145/3337167.3337175. URL `https://doi.org/10.1145/3337167.3337175`.

[170] Brad Sherman. Intangible machines: Patent protection for software in the United States. *History of Science*, 57(1):18–37, 2019. doi: 10.1177/0073275318770781.

[171] Laura L. Sherman, Arthur C. Taylor III, Larry L. Green, Perry A. Newman, Gene W. Hou, and Vamshi Mohan Korivi. First- and Second-Order Aerodynamic Sensitivity Derivatives via Automatic Differentiation with Incremental Iterative Methods. *Journal of Computational Physics*, 129(2):307–331, 1996. ISSN 0021-9991. doi: https://doi.org/10.1006/jcph.1996.0252. URL `https://www.sciencedirect.com/science/article/pii/S0021999196902521`.

[172] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons (Asia) Pte Ltd, 8th edition, international student version edition, 2010.

[173] Are Strandlie and Rudolf Frühwirth. Track and vertex reconstruction: From classical to adaptive methods. *Reviews of Modern Physics*, 82(2):1419–1458, 2010. ISSN 0034-6861, 1539-0756. doi: 10.1103/RevModPhys.82.1419. URL `https://link.aps.org/doi/10.1103/RevModPhys.82.1419`.

[174] Ganesh Tambave, Johan Alme, Gergely Gábor Barnaföldi, Rene Barthel, Anthony van den Brink, Stephan Brons, Mamdouh Chaar, Viljar Eikeland, Georgi Genov, Ola Grøttvik, Helge Egil Seime Pettersen, Željko Pastuović, Simon Huiberts, Håvard Helstrup, Kristin Fanebust Hetland, Shruti Mehendale, Ilker Meric, Qasim Waheed Malik, Odd Harald Odland, Gábor Papp, Thomas Peitzmann, Pierluigi Piersimoni, Attiq Ur Rehman, Felix Reidt, Matthias Richter, Dieter Röhrich, Ákos Sudár, Andreas Tefre Samnøy, Joao Seco, Hesam Shafiee, Eivind Vågslid Skjæveland, Jarle Rambo Sølie, Kjetil Ullaland, Mónika Varga-Kőfaragó, Lennart Volz, Boris Wagner, and Shiming Yang. Characterization of monolithic CMOS pixel sensor chip with ion beams for application in particle computed tomography. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 958(Proceedings of the Vienna Conference of Instrumentation 2019):162626, April 2020. ISSN 01689002. doi: 10.1016/j.nima.2019.162626.

[175] Raymond J. Toal. System Calls. URL `https://cs.lmu.edu/~ray/notes/syscalls/`.

[176] Markus Towara and Uwe Naumann. A Discrete Adjoint Model for OpenFOAM. *Procedia Computer Science*, 18:429–438, 2013. ISSN 18770509. doi: 10.1016/j.pr

ocs.2013.05.206. URL https://linkinghub.elsevier.com/retrieve/pii/S1877050913003499.

[177] Markus Towara, Michel Schanen, and Uwe Naumann. MPI-Parallel Discrete Adjoint OpenFOAM. *Procedia Computer Science*, 51:19–28, 2015. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2015.05.181. URL https://www.sciencedirect.com/science/article/pii/S1877050915009898. International Conference On Computational Science, ICCS 2015.

[178] Jean Utke. Flattening Basic Blocks. In Martin Bücker, George Corliss, Uwe Naumann, Paul Hovland, and Boyana Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, pages 121–133, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-28438-3.

[179] Vassil Vassilev, Martin Vassilev, Alexander Penev, Lorenzo Moneta, and Violeta Ilieva. Clad — Automatic Differentiation Using Clang and LLVM. *Journal of Physics: Conference Series*, 608:012055, 2015. doi: 10.1088/1742-6596/608/1/012055. URL https://doi.org/10.1088/1742-6596/608/1/012055.

[180] Delio Vicini, Sébastien Speierer, and Wenzel Jakob. Differentiable signed distance function rendering. *ACM Transactions on Graphics*, 41(4), jul 2022. ISSN 0730-0301. doi: 10.1145/3528223.3530139. URL https://doi.org/10.1145/3528223.3530139.

[181] Tanya Volnina. Enable AVX-512 instructions in Valgrind. Free and Open source Software Developers' European Meeting (FOSDEM), February 2022. URL https://archive.fosdem.org/2022/schedule/event/valgrind_avx512/.

[182] John von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. doi: 10.1109/85.238389.

[183] Jochen Voss. When can we interchange the derivative with an expectation? Mathematics Stack Exchange. URL https://math.stackexchange.com/q/1986477. URL:https://math.stackexchange.com/q/1986477 (version: 2020-03-18).

[184] Andrea Walther and Andreas Griewank. Getting started with ADOL-C. In Uwe Naumann and Olaf Schenk, editors, *Combinatorial Scientific Computing*, chapter 7, pages 181–202. Chapman-Hall CRC Computational Science, 2012.

[185] Liang-Kai Wang, Charles Tsen, Michael J. Schulte, and Divya Jhalani. Benchmarks and performance analysis of decimal floating-point applications. In *2007 25th International Conference on Computer Design*, pages 164–170, 2007. doi: 10.1109/ICCD.2007.4601896.

[186] Wujie Wang, Simon Axelrod, and Rafael Gómez-Bombarelli. Differentiable Molecular Simulations for Learning and Control. In *Machine Learning for Molecules Workshop at NeurIPS 2020*, 2020.

[187] Wujie Wang, Zhenghao Wu, Johannes C. B. Dietschreit, and Rafael Gómez-Bombarelli. Learning pair potentials using differentiable simulations. *The Journal of Chemical Physics*, 158(4):044113, January 2023. ISSN 0021-9606. doi: 10.1063/5.0126475. URL `https://doi.org/10.1063/5.0126475`.

[188] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *Proceedings of the 4th International Conference on Computational Science (ICCS 2004)*, June 2004.

[189] R. E. Wengert. A Simple Automatic Derivative Evaluation Program. *Communications of the ACM*, 7(8):463–464, August 1964. ISSN 0001-0782. doi: 10.1145/355586.364791. URL `https://doi.org/10.1145/355586.364791`.

[190] Robert R. Wilson. Radiological use of fast protons. *Radiology*, 47(5):487–491, 1946. ISSN 0033-8419, 1527-1315. doi: 10.1148/47.5.487. URL `http://pubs.rsna.org/doi/10.1148/47.5.487`.

[191] WIPO Copyright Treaty. Adopted in Geneva on December 20, 1996.

[192] Patrick Wohlfahrt and Christian Richter. Status and innovations in pre-treatment CT imaging for proton therapy. *The British Journal of Radiology*, 93(1107): 20190590, March 2020. ISSN 0007-1285, 1748-880X. doi: 10.1259/bjr.20190590.00004.

[193] World Health Organization. WHO report on cancer: setting priorities, investing wisely and providing care for all. 2020. URL `https://apps.who.int/iris/rest/bitstreams/1267643/retrieve`.

[194] World Intellectual Property Organization. What is intellectual property? 2020.

[195] Hao Xie, Jin-Guo Liu, and Lei Wang. Automatic differentiation of dominant eigensolver and its applications in quantum physics. *Physical Review B*, 101: 245139, June 2020. doi: 10.1103/PhysRevB.101.245139. URL `https://link.aps.org/doi/10.1103/PhysRevB.101.245139`.

[196] Ming Yang, Gary Virshup, James Clayton, Xiaorong Ronald Zhu, R Mohan, and Lei Dong. Theoretical variance analysis of single- and dual-energy computed tomography methods for calculating proton stopping power ratios of biological tissues. *Physics in Medicine and Biology*, 55(5):1343–1362, 2010. ISSN 0031-9155, 1361-6560. doi: 10.1088/0031-9155/55/5/006. URL `https://iopscience.iop.org/article/10.1088/0031-9155/55/5/006`.

[197] Ming Yang, X Ronald Zhu, Peter C Park, Uwe Titt, Radhe Mohan, Gary Virshup, James E Clayton, and Lei Dong. Comprehensive analysis of proton range uncertainties related to patient stopping-power-ratio estimation using the stoichiometric calibration. *Physics in Medicine and Biology*, 57(13):4095–4115, June 2012. doi: 10.1088/0031-9155/57/13/4095. URL `https://doi.org/10.1088/0031-9155/57/13/4095`.

[198] Toshio Yoshida. SPARC64 X+ : Fujitsu's next generation processor for the UNIX servers, 2013. URL `https://hc25.hotchips.org/`. Presentation at Hot Chips 25, August 25-27, 2013, Stanford.

[199] Wencheng Zhang, Xiaodong Zhang, Pei Yang, Pierre Blanchard, Adam S. Garden, Brandon Gunn, C. David Fuller, Mark Chambers, Katherine A. Hutcheson, Rong Ye, Stephen Y. Lai, Mohamed Abdallah Sherif Radwan, X. Ron Zhu, and Steven J. Frank. Intensity-modulated proton therapy and osteoradionecrosis in oropharyngeal cancer. *Radiotherapy and Oncology*, 123(3):401–405, 2017. ISSN 0167-8140. doi: https://doi.org/10.1016/j.radonc.2017.05.006. URL `https://www.sciencedirect.com/science/article/pii/S0167814017303699`.

[200] Hongqing Zhu, Huazhong Shu, and Meiyu Ding. Numerical solutions of two-dimensional Burgers' equations by discrete Adomian decomposition method. *Computers & Mathematics with Applications*, 60(3):840–848, 2010. ISSN 0898-1221. doi: https://doi.org/10.1016/j.camwa.2010.05.031. URL `https://www.sciencedirect.com/science/article/pii/S0898122110003883`.

[201] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi: 10.1109/TIT.1977.1055714.

Academic Curriculum Vitae

Max Aehle studied mathematics at Technische Universität Kaiserslautern, with a minor in mechanical engineering.

He received his Bachelor's degree with Prof. Dr. Andreas Gathmann in the Algebra, Geometry and Computer Algebra group in September 2018. His Master's thesis on the *simulation of gas pipelines via asymptotic expansions* was jointly supervised by PD Dr. Raul Borsche in the Industrial Mathematics group, and Dr. Jan Mohring at the Fraunhofer Institute for Industrial Mathematics.

Since October 2020, he works as a research assistant at the Chair for Scientific Computing under the supervision of Prof. Dr. Nicolas R. Gauger, mainly funded by the SIVERT research training group. His research interests are focused on algorithmic differentiation, dynamic binary instrumentation, scientific computing, and applications in computational fluid dynamics and high-energy physics.