Forward-Mode Automatic Differentiation of Compiled Programs

MAX AEHLE, JOHANNES BLÜHDORN, MAX SAGEBAUM, and NICOLAS R. GAUGER, University of Kaiserslautern-Landau (RPTU), Germany

Algorithmic differentiation (AD) is a set of techniques that provide partial derivatives of computer-implemented functions. Such a function can be supplied to state-of-the-art AD tools via its source code, or via an intermediate representation produced while compiling its source code.

We present the novel AD tool Derivgrind, which augments the machine code of compiled programs with forward-mode AD logic. Derivgrind leverages the Valgrind instrumentation framework for a structured access to the machine code, and a shadow memory tool to store dot values. Access to the source code is required at most for the files in which input and output variables are defined.

Derivgrind’s versatility comes at the price of scaling the run-time by a factor between 30 and 75, measured on a benchmark based on a numerical solver for a partial differential equation. Results of our extensive regression test suite indicate that Derivgrind produces correct results on GCC- and Clang-compiled programs, including a Python interpreter, with a small number of exceptions. While we provide a list of scenarios that Derivgrind does not handle correctly, nearly all of them are academic counterexamples or originate from highly optimized math libraries. As long as differentiating those is avoided, Derivgrind can be applied to an unprecedentedly wide range of cross-language or partially closed-source software with little integration efforts.

CCS Concepts: • Mathematics of computing → Automatic differentiation; • Software and its engineering → Runtime environments; Dynamic analysis.

Additional Key Words and Phrases: Algorithmic Differentiation, Differentiable Programming, Dynamic Binary Instrumentation, Valgrind, Derivgrind

1 INTRODUCTION

In many areas of science and technology, processes of interest can be described by a function \( f : \mathbb{R}^n \rightarrow \mathbb{R}^m, x \mapsto y \) implemented in a computer program. In order to leverage such a simulation for, e.g., gradient-based optimization, knowledge of the derivatives \( \frac{\partial y}{\partial x} \) is required. To this end, besides numerical and symbolic differentiation, algorithmic differentiation (AD) [Griewank and Walther 2008] has found widespread use in various areas such as aerodynamic shape optimization [Luers et al. 2018] and machine learning [Baydin et al. 2017]. AD is a set of techniques to turn a computer program for \( f \) into a floating-point accurate computer program for \( \frac{\partial y}{\partial x} \). Both the forward and reverse mode of AD run the original program \( f \), and perform additional AD logic before, alongside, and afterwards. This work is concerned with the forward mode in the case of a single \((n = 1)\) input variable \( x \): For each input, intermediate, and output value \( a \), the AD logic additionally computes its dot or tangent value \( \dot{a} = \frac{\partial f}{\partial x} \). The dot value \( \dot{x} \) of the single AD input is initialized with 1, \( \dot{a} \) is initialized with 0 for every other global or constant variable \( a \), and each real arithmetic operation \( a_{\text{lhs}} = \phi(a_1, \ldots, a_k) \) performed by the original program is matched by an update

\[
\frac{\partial f}{\partial a_1} \cdot \dot{a}_1 + \cdots + \frac{\partial f}{\partial a_k} \cdot \dot{a}_k.
\]
For instance, a multiplication $a_{\text{lhs}} = a_1 \cdot a_2$ leads to an update $\dot{a}_{\text{lhs}} = a_2 \cdot \dot{a}_1 + a_1 \cdot \dot{a}_2$ of the dot value of the product, and $a_{\text{lhs}} = \sin(a_1)$ should be accompanied by $\dot{a}_{\text{lhs}} = \cos(a_1) \cdot \dot{a}_1$. Several ways for the augmentation of the program with AD logic such as (1) have been described in the literature.

- **Source transformation** tools like TAPENADE [Hascoet and Pascual 2013] parse the source code of the program; forward-mode AD logic like (1) can then be inserted directly into the source code.

- **Operator overloading** tools like ADOL-C [Walther and Griewank 2012], CoDiPack [Sagebaum et al. 2019] and autograd [Maclaurin et al. 2015] rely on a language feature of many object-oriented languages like C++ and Python. They define a special type to be used instead of the built-in floating-point types such as double and np.array. The new type stores both $a$ and additional AD data such as $\dot{a}$, and overloads all elementary operations and math functions with their original action on $a$ accompanied by additional AD logic.

- AD logic can also be added during compilation. The AD tool CLAD [Vassilev et al. 2015] is a plugin for the Clang front-end of the LLVM compiler infrastructure, operating on Clang’s internal representation of the C/C++ source code in the form of an abstract syntax tree (AST). Enzyme [Moses and Churavy 2020] operates on the LLVM intermediate representation (IR).

Except for Enzyme, a common weakness of these approaches is that they rely on the entire source code implementing the real arithmetic to be differentiated. Thus, they cannot be applied if these sources are not fully available, or if they are written in an exotic programming language, or a mix of languages for which no AD tool exists yet.

Different languages are not a problem for Enzyme as long as an LLVM front-end exists for all of them. Also, Enzyme makes it possible to include functions from static libraries into AD without need for their source code, if these static libraries were shipped with embedded LLVM IR.

In this work, we go even further away from the source code. Our novel tool Derivgrind\footnote{https://www.scicomp.uni-kl.de/software/derivgrind/ and https://github.com/SciCompKL/derivgrind} propagates dot values through a compiled “client” program, dynamically augmenting portions of its machine code with forward-mode AD logic just before they execute on the processor. Users of Derivgrind only need access to a (usually small) number of source files in which they want to define AD input and output variables. Derivgrind therefore enables the integration of forward-mode AD into complex, cross-language software projects involving closed-source dependencies, with minimal effort. We target clients running under Linux on both the x86 (also referred to as i386 or IA-32) and amd64 (x86-64, Intel 64 or x64) instruction set architectures.

Derivgrind has been implemented as a tool within the Valgrind framework for building dynamic analysis tools [Nethercote and Seward 2007b; Seward et al. 2022]. After invoking

```
valgrind --tool=⟨tool⟩ ⟨tool options⟩ ⟨client executable⟩ ⟨arguments for client⟩
```

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. Valgrind translates these portions into the unified object-oriented VEX IR, which it presents to the selected Valgrind tool. The tool can modify (“instrument”) the VEX IR. Afterwards, the Valgrind core translates the instrumented VEX code back to machine code, which can then be executed on the CPU; we may think of this as running the instrumented VEX code on a “synthetic CPU”.

As an example, Valgrind’s default tool Memcheck [Seward and Nethercote 2005] adds instrumentation to keep track of so-called availability and validity bits and detect memory errors based on this information. A standard Valgrind distribution includes several other tools for thread error detection and profiling. Additional tools have been created independently, e. g. for taint analysis [Drewry and Ormandy 2007; Newsome and Song 2015], crash consistency checking of persistent memory
applications [Jenkins and Scott 2020; Kapela 2015], or the analysis of floating-point accuracy problems [Benz et al. 2012; Févotte and Lathuilière 2017, 2019; Grasland et al. 2019; Sanchez-Stern et al. 2018]. Our novel tool Derivgrind augments the VEX IR with forward-mode AD logic.

Besides the instrumentation, Valgrind tools can specify monitor commands and client requests to enable communication between the tool, the user, and the client program. Derivgrind uses this feature for setting and getting dot values. As Valgrind tools can wrap library functions, Derivgrind provides analytical derivatives for functions from the C math library.

This paper is structured as follows. In Section 2, we recall how compiled programs usually handle floating-point data, and how this handling can be extended to propagate dot values alongside. Section 3 starts with a review of VEX IR, and describes the specific AD instrumentation added by Derivgrind. Sections 4 and 5 detail the monitor commands and client requests, and the function wrappers introduced by Derivgrind. We apply Derivgrind in multiple case studies for validation and performance analysis in Section 6, and close with a summary and outlook in Section 7.

2 INSTRUMENTATION OF MACHINE CODE WITH FORWARD AD LOGIC

2.1 Shadowing Approach

Our approach to store dot values is to match every storage location, such as memory and registers, by a shadow storage location of the same size and type. Whenever a location stores a representation of a floating-point value \( a \) in any format, the AD instrumentation should make sure that the corresponding shadow location stores the dot value \( \dot{a} \) in the same format. This does also apply to individual bytes of a floating-point representation in case the representation is split up into several parts, e.g. to copy them separately. When the data at a location is not part of a binary representation of a floating-point value, the content of the corresponding shadow location is unspecified.

2.2 Floating-Point Formats and Instructions

Among other representations of real numbers as digital data, the IEEE-754 standard [IEEE 2008], revised in 2008, specifies the most commonly used binary floating-point interchange formats binary32 (formerly: single) and binary64 (formerly: double). 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,

so \( 0b\beta_63\beta_62...\beta_1\beta_0 \) represents the real number

\[
(-1)^{\beta_{63}} \cdot \left( 1 \cdot 2^E + \beta_{51} \cdot 2^{E-1} + \beta_{50} \cdot 2^{E-2} + \cdots + \beta_0 \cdot 2^{E-52} \right)
\]

with \( E = \beta_{62} \cdot 2^{10} + \beta_{61} \cdot 2^9 + \cdots + \beta_{52} \cdot 2^0 - 1023 \).

A different formula applies if the exponent is 0b00...00, to represent numbers close to zero without an implicit leading digit 1. In particular, a 64-bit 0b00...00 is interpreted as +0.0. The exponent 0b11...11 is used to represent infinite numbers and not-numbers (NaNs). 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.

On x86 and amd64 CPUs, real arithmetic is provided by the floating-point instruction sets x87 and SSE. Most of these instructions expect real numbers to be represented as binary32 or binary64,
or a sequence of those in a single-instruction-multiple-data (SIMD) vector [AMD64 Technology 2021]. Compilers typically use binary32 to implement the C/C++ type float and the Fortran type real, and binary64 for double and double precision.

Internally, the x87 floating-point registers use an 80-bit double-extended precision format. While the x87 instructions flds/fstps and f1dl/fstpl convert from/to binary32 and binary64 when they move data between the floating-point registers and memory, the instructions fldt/fstp t expose the 80-bit format to memory. The GCC and Clang compilers use the 80-bit x87 format to implement the C/C++ type long double.

Our main assumption on the client program is that it performs real arithmetic only by the corresponding floating-point instructions (with the exceptions listed below). To apply forward-mode AD, these instructions have to be recognized and augmented with instructions acting on the operands and their dot values according to basic rules of differential calculus.

The client program may, moreover, use type-agnostic instructions to move floating-point values, even in several portions like single bytes. In most cases, these instructions are simply applied to the dot values as well. Compare-and-swap (CAS) instructions need special consideration in Section 2.3.

During our work, we encountered a variety of tricky ways to perform real arithmetic using integer or bitwise logical instructions, or by a “misuse” of floating-point instructions. The most important of these bit-tricks, presented in Section 2.4, can be dealt with by a suitable instrumentation of bitwise logical instructions. Further scenarios listed in Section 2.5 are not covered by our current implementation.

2.3 Compare-And-Swap Instructions

A CAS instruction like lock cmpxchqg makes a copy of the present value at a memory location addr, compares it with an “expected” value expd, and only if they match, writes another given value new to addr. The copy of the previous value at addr can be used to determine whether the write to addr took place. CAS instructions are useful in setups with multiple threads accessing a shared state, because all of their operations happen in a single “atomic” step. Threads can use CAS instructions to make sure in a thread-safe way that when they update the shared state at addr, it has not changed while the update was computed. Otherwise, if a thread $T_1$ used a non-atomic conditional write and another thread $T_2$ wrote to addr after the comparison but before the write of $T_1$, the update by $T_2$ would be discarded.

For the correct AD handling of CAS instructions, it is important to realize that the shared state of the augmented program consists of both the value at addr, and the dot value in the 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 yield equality, 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.

2.4 Bitwise Logical Instructions

A bitwise logical “and” operation between a binary32/binary64 and $0b01\ldots11$ sets the sign bit to zero, and thus computes the absolute value. Listing 1 demonstrates that GCC routinely uses this “bit-trick”. The assembly code on the right has been produced with GCC 11.2.0 from the C code on the left, using the flags -S and -O3. It was stripped from irrelevant labels and directives, and is explained in the following. By the System V ABI [Matz et al. 2012], the floating-point argument of
Forward-Mode Automatic Differentiation of Compiled Programs

Listing 1. 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.

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

the function f is passed in the lower half of the 128-bit register XMM0. The four four-byte constants 0b11...11, 0b01...11, 0b00...00 and 0b00...00, inserted by the .long directives, compose the other 128-bit operand to andpd. Note that the most significant byte 0b01111111 of the second four-byte block is stored at the highest memory address within that block, as x86 and amd64 stick to the little-endian storage order. Therefore, its zero bit aligns with the sign bit of the binary64 in the lower half of XMM0.

As a consequence, the AD instrumentation of an “and” instruction has to inspect both operands x and y. If x is equal to 0b01...11, y might represent a real number whose absolute value is taken by this instruction, and the AD instrumentation must act according to the respective differentiation rule. Analogous considerations apply vice versa. In case both operands of an “and” instruction are 0b01...11, no differentiable real arithmetic is performed because 0b01...11, as a binary32 or binary64, represents not-a-number.

To handle SIMD operations correctly, the search for 0b01...11 must be performed on all 32-bit and 64-bit blocks of the operands.

A similar procedure has to be applied for the bitwise logical “or” and “exclusive-or” instructions when one operand is 0b10...00, as these operations can compute the negative absolute value or negative value, respectively, of a binary32 or binary64 in the other operand. As an additional complication, 0b10...00 represents the real number −0.0. Therefore, taking the negative of −0.0 can lead to an “exclusive-or” instruction whose operands are both 0b10...00, making it ambiguous which operand represents the real number, and which dot value the differentiation rule of negation should thus be applied to. To resolve this issue, we interpret a logical “or” or “exclusive-or” as an arithmetic operation only if the dot value of the operand 0b10...00 is 0x00...00.

A second important way in which floating-point arguments pass through bitwise logical instructions is given by the masking pattern

```
(mask and iftrue) or ((not mask) and iffalse).
```

The result of this expression is assembled in a bitwise fashion from iftrue and iffalse depending on whether the respective bit in the mask is 1 or 0. We observed this pattern being used by Clang 14.0.0 for the code in Listing 2. In this example, the mask is created by the cmpltsd instruction, which sets XMM2 to either 0xff...ff if a < 0, or 0x00...00 otherwise. The subsequent “and”, “and-not” and “or” instructions then place in XMM2 either the constant 2.0 copied from .LCPI0_0, or the value a copied from XMM0, respectively. In both cases, finally XMM2 is added to the register XMM0, which holds the input a in the beginning, and the return value at the end (realizing 2 · a as a + a in the “else” branch).
Listing 2. Clang 14.0.0 may use logical operations in a masking pattern to select one of two floating-point numbers.

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

We support masking patterns as long as entire binary32s or binary64s are picked in the style of an if-then-else operation, as in the previous example. To this end,

- if one operand of a bitwise logical “and” is 0xff...ff, or
- if one operand of a bitwise logical “or” is 0x00...00 and has the dot value 0x00...00,

the dot value of the expression is the dot value of the other operand.

2.5 Unhandled Hidden Floating-Point Arithmetics

Unfortunately, we are not aware of any comprehensive approach to systematically detect all the real arithmetic “hidden” in a portion of machine code. This subsection lists scenarios in which our current implementation fails to produce the correct dot values, and indicates patterns that programmers and compilers should avoid to make the machine code “AD-friendly”.

**Masking of incomplete floating-point representations.** The approach of Section 2.4 to handle masking patterns breaks down if a binary32 or binary64 is not entirely selected, e.g. in order to exchange the exponent bits to implement the C math function `frexp`.

**Integer additions to the exponent.** Multiplications with powers of two can be implemented by integer additions to the exponent bits, possibly in conjunction with bit-shifts. We observed this pattern in the implementation of the exponential function for a binary32 SIMD vector in NumPy, where initially a multiple $k \cdot \ln 2$ is subtracted from the argument to map it into $[0, \ln 2]$, and the result is scaled by $2^k$ to account for this range reduction$^2$. It is difficult to decide which of the two summands represents the floating-point number.

**Emulation of floating-point instructions.** As a generalization of the previous items, floating-point libraries like GNU MPFR [Fousse et al. 2007] emulate arithmetic operations of standardized or proprietary floating-point formats by integer and bitwise logical instructions. The decimal floating-point arithmetic upcoming with the C 23 language standard must, as long as there is no hardware support for the IEEE-754 formats decimal32, decimal64, decimal128, also be implemented in software$^3$.

---

$^2$NumPy [Harris et al. 2020] version 1.19.5, https://github.com/numpy/numpy/blob/8f4b73a0d04f7beb06a154b43e5ef5b5980052f/numpy/core/src/umath/simd.inc.src#L1558

$^3$GCC 11.2 does it like this: https://github.com/gcc-mirror/gcc/blob/b454e40956947938c9e274d75ce8a43171f3efa/libgcc/config/libbid/bid64_add.c
**Instruction sequences composing a binary identity.** When a client program passes around real numbers as decimal strings or in an otherwise encoded or encrypted form, this obviously erases the dot values. GCC’s implementation of an OpenMP atomic addition to a double on the x86 Pentium CPU initially copies the original value on the stack using the instructions `fldq`, `fistpq`, i.e. reinterprets the value as an integer, converts it to a 80-bit floating-point number, and then converts it back\(^4\).

**Rational arithmetic.** As integer operations are ignored by our approach, any calculations with integer fractions are not recognized as performing real arithmetic.

**Exploiting floating-point imprecision for rounding.** In order to represent a real number \(x\) with \(2^{52} \leq |x| < 2^{53}\) in the binary64 format according to formula (2), the exponent \(E\) has to be chosen as 52, so the least-significant bit of the significand controls the binary digit \(2^{E-52} = 1\). Therefore within the above range for \(x\), the real numbers representable by binary64 are precisely the integral numbers. The following procedure exploits this fact to round a binary64 \(y\) with \(|y| < 2^{51}\) to an integral number. 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 floating-point addition’s rounding mode. Immediately subtracting \(T\) again does not introduce any more floating-point errors, so we end up with the value of \(y\) rounded to an integral number.

The GLIBC math library uses tricks of this kind in several places, e.g. for range reduction (shifting the argument to \(\sin\) into \([-\frac{\pi}{4}, \frac{\pi}{4}]\) by adding a multiple of \(\frac{\pi}{4}\))\(^5\) or to compute an index into a lookup table\(^6\). While the correct dot value of the result of a rounding operation is zero, the instrumented code, according to the above analytical differentiation rules, adds and immediately subtracts \(\dot{T} = 0\) from \(\dot{y}\), leaving it unchanged because there are no floating-point errors.

Note that dangerous constructs in the C math library are not problematic for Derivgrind if math function wrappers (Section 5) are used.

**Summary and comment.** As real arithmetic can be hidden in machine code in various ways, it would be unrealistic to aim for a universal “AD tool for machine code” supporting each and every possible client program. Instead, we require that the authors of the client program and the applied compilers stick to the binary32, binary64, and x87 80-bit format to represent real numbers. Furthermore, we assume that they perform real arithmetic only by the corresponding floating-point instructions, by calls to functions from the C math library, and by the supported bit-tricks listed in Section 2.4. From our collection of regression tests described in Section 6, only a small number violate this assumption and fail. Therefore, we believe that our assumption is only a mild limitation regarding a productive use of AD for compiled programs.

### 3 IMPLEMENTATION OF AD INSTRUMENTATION IN DERIVGRIND

Instead of directly working with the machine code of the client program, we have implemented the forward-mode AD instrumentation of Section 2 using the Valgrind framework [Nethercote and Seward 2007b; Seward et al. 2022]. Therefore, our tool Derivgrind operates on Valgrind’s object-oriented internal representation of machine code, VEX IR, in portions called superblocks.

\(^4\)GCC 11.2 with \(-m32\) \(-march=pentium \) \(-O3\), https://godbolt.org/z/hc8ehfG88
\(^5\)GLIBC version 2.35, https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/ieee754/dbl-64/s_sin.c;h=8e65f7cc00faf64f4c a85a2a1e937280bcd06d3a;hb=HEAD\#l156
\(^6\)GLIBC version 2.35, https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/ieee754/dbl-64/s_sin.c;h=8e65f7cc00faf64f4c a85a2a1e937280bcd06d3a;hb=HEAD\#l136
Listing 3. Example of the textual representation of a VEX IR superblock, from the documentation of Valgrind. Every line represents a statement. We have emphasized parameters with red text and expressions with a blue frame.

```
------ 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
```

In Section 3.1, we revisit the necessary details of VEX. More extensive documentation can be found in the source code of Valgrind\(^7\). Section 3.2 describes how shadow locations for dot values are obtained, and Sections 3.3 and 3.4 detail how Derivgrind instruments the two main building blocks of VEX to process the dot values. Finally, we remark on system calls in Section 3.5 and on limitations of Valgrind in Section 3.6.

### 3.1 Basic Structure of VEX IR

**Synthetic CPU.** As VEX IR strives to be independent from hardware and instruction set architectures, 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. In addition, VEX IR provides temporaries to store intermediate values for the scope of the current superblock. Temporaries are specified by an index and can be assigned only once.

**Statements and expressions.** Each superblock contains a list of statements, which represent actions with side effects, such as those listed in the left column of Table 1. Depending on the type of action, a statement involves further parameters and expressions. A parameter is a value defined during instrumentation, and is denoted by angle brackets \( ⟨·⟩ \) in Tables 1 and 2. An expression represents a value to be computed when the containing statement is executed on the synthetic CPU, without side effects. We denote expressions by italic text in Tables 1 to 3. The left column of Table 2 lists types of expressions along with the parameters and sub-expressions that they involve. A very important type of expressions acquire their value on the synthetic CPU by an (integer arithmetic, floating-point arithmetic, bitwise logical, ...) operation applied to one to four sub-expressions. We provide more examples of those in the left column of Table 3.

Listing 3 reproduces an example of a VEX IR superblock’s textual representation from the documentation of Valgrind, corresponding to the 4-byte integer addition of the general-purpose register EAX to EBX on x86. Each of the five lines represents one statement. The first statement is meta information, the next three statements write to temporaries and the last statement writes to the guest state. Expressions (indicated by a blue frame) specify which data is written. For the first two writes, the GET expression represents reads from the registers with byte offsets 0 and 12. The expression on the right hand side of the fourth line applies an operation. Its operands are expressions themselves, which acquire their value by reading from the temporaries with indices 3 and 2. The right hand side of the fifth line reads from temporary 1.

---

\(^7\)https://sourceware.org/git/?p=valgrind.git;a=blob;f=VEX/pub/libvex_ir.h;h=85805bb69b8b447d0d5cd362b1fd3e5b9b181c28;hb=8b2cf214afb2590cecf5ff7cabeb6cecc3862ade
Dirty calls and CCalls. VEX achieves its full generality through dirty call statements and CCall expressions. Both store a name, a function pointer and a tuple of expressions (among other things), and when execution on the synthetic CPU reaches the dirty call or CCall, the stored function is called with arguments taken from the evaluated expressions. The Valgrind core represents certain x86 and amd64 instructions by dirty calls when there is no architecture-independent equivalent in VEX.

For example, the x87 instruction `fldt` loads 10 bytes of memory into an x87 floating-point register, which internally uses 80 bits to represent a floating-point number. The other way round, `fstp` stores an 80-bit floating-point representation in memory. Note that VEX IR does not support 80-bit arithmetic. Instead, Valgrind represents the x87 floating-point registers by `binary64` s in the guest state of the synthetic CPU, and replaces 80-bit by 64-bit arithmetic. Therefore, dirty calls replacing the `fldt` and `fstp` instructions have to convert between the `binary64` format in the guest state and the 80-bit format in memory.

Additionally, Derivgrind emits dirty calls to access the shadow memory from within VEX, and CCalls for the rather complicated AD augmentation of bitwise logical instructions (Section 2.4).

3.2 Shadow Temporaries, Registers and Memory
As described in Section 2.1, Derivgrind provides a shadow location to every storage location.

Temporaries. Within each superblock, Derivgrind shadows every temporary \( i \) with the temporary \( (i + m_{\text{tmp}}) \), using a shift \( m_{\text{tmp}} \) larger than the maximal index of a temporary before instrumentation. When Derivgrind needs additional temporaries to compute dot values, it allocates them with indices greater than or equal to \( 2 \cdot m_{\text{tmp}} \).

Registers. Before instrumentation, the VEX code uses a known, architecture-dependent number \( m_{gs} \) of bytes of the guest state. Valgrind’s synthetic CPU provides \( 3 \cdot m_{gs} \) bytes of guest state to the instrumented code. For each register with byte offset \( j \), Derivgrind can therefore use the “shadow register” with byte offset \( j + m_{gs} \) to store the dot value.

Memory. The address shifting approach used to shadow temporaries and registers is not feasible for memory, as the client program can use addresses throughout the whole virtual address space, and the tool has limited control about which parts of the virtual address space it might itself make allocations in. Like Memcheck, Derivgrind therefore uses a shadow memory tool, developed by us\(^8\) according to Nethercote and Seward [2007a]. It can be thought of as a big hashmap assigning the content of shadow memory to memory addresses; the actual implementation uses a prefix tree (trie) data structure similar to multilevel page tables for virtual memory. As the default value of an uninitialized byte in the shadow memory is 0x00, `binary32` s and `binary64` s in the data and bss segment of the client program (such as C/C++ floating-point variables with static storage duration) have the correct initial dot value +0.0.

3.3 Wrapping Statements
For all statements with relevance to AD and except for CAS statements, Derivgrind inserts the forward-mode AD logic (1) in the form of a “differentiated statement” in front of the original statement. Examples of such differentiated statements are shown in Table 1. Most of them involve “differentiated expressions” that compute the dot value of a value computed by an expression in the original statement. Differentiated expressions have been marked with a dot, and can be formed as described in Section 3.4.

\(^8\)https://github.com/SciCompKL/derivgrind
Table 1. VEX IR statements, and their augmentation with forward-mode AD logic.

| VEX statement                                      | Differentiated VEX statement                                      |
|---------------------------------------------------|-------------------------------------------------------------------|
| Write data to a temporary with index $i$.         | Write differentiated data to the shadow temporary.               |
| $t(i) = p$                                        | $t(i + m_{\text{tmp}}) = \dot{p}$                                |
| Write data to the register with byte offset $j$ in | Write differentiated data to the shadow register.                |
| the guest state.                                  | $\text{PUT}(\langle j \rangle) = p$                              |
| $\text{PUT}(\langle j \rangle + m_{\text{gs}}) = \dot{p}$ |                                                                   |
| Write data to memory.                             | Write differentiated data to shadow memory.                      |
| $\text{STle}(\text{address}) = p$                 | Implemented as a dirty call to access the shadow memory from VEX.|
| Write data to memory, if a condition is satisfied.|                                                                   |
| if (guard) $\text{STle}(\text{address}) = p$     | Write differentiated data to shadow memory, subject to the same  |
|                                                   | condition. Implemented as a dirty call.                          |
| Compare-and-swap, loading $addr$ into temporary    | CAS statements are replaced as discussed in Section 2.3.         |
| $t(\text{old})$, and replacing data at $addr$ by $new$ |                                                                   |
| $t(\text{old}) = \text{CASle}(addr :: \text{expd} \rightarrow new)$ | The augmentation depends on the dirty call, see details in Section 3.3.|
| Dirty call, invoking a Valgrind function with side | The augmentation depends on the dirty call, see details in Section 3.3.|
| effects.                                         |                                                                   |
| ------ IMark(...) ------                          | Not relevant for AD.                                             |
| Conditional jump.                                 | Not relevant for AD.                                             |
| if (guard) goto {<jump kind>} (target)            |                                                                   |

Derivgrind replaces a CAS statement by an entirely new sequence of statements, in order to perform the additional check whether the dot value has changed, as discussed in Section 2.3. Dirty call statements can be identified by their name. Derivgrind matches dirty calls emitted for `fldt` and `fstpt` (see Section 3.1) by dirty calls performing the same operation on the shadow guest state and shadow memory, to comply with our convention to store the dot value of every floating-point variable in the same format as its value (Section 2.1).

As far as we observed it, all the other dirty calls emitted by the Valgrind core from x86 and amd64 machine code can hardly be part of a floating-point calculation. For example, they perform actions such as obtaining information about the CPU hardware (the VEX IR equivalent of the x86 instruction `cpuid`), reading a time-stamp counter (`rdtsc`), SSE 4.2 string operations (`pcmp(<X> str <Y>)`, or (re)storing some SSE status bits. These dirty calls do not require any additional AD logic, except that shadows of output temporaries should be assigned some value so they are initialized in case they are later read from.

### 3.4 Wrapping Expressions

Many of the differentiated statements (Section 3.3 and Table 1) involve a differentiated expression $\dot{p}$ computing the dot value of the result of an expression $p$ in the original statement. Derivgrind
Table 2. VEX IR expressions, and their forward-mode algorithmic derivatives.

| Expression                  | Differentiated expression |
|-----------------------------|---------------------------|
| Read from a temporary with index $i$. $t(i)$ | Read from the shadow temporary. $t(i + m_{tmp})$ |
| Read data of specified type from the register with byte offset $j$ in the guest state. $\text{GET:} (\text{type})(\langle j \rangle)$ | Read data of the same type from the shadow register. $\text{GET:} (\text{type})(\langle j + m_{gs} \rangle)$ |
| Read from memory. $\text{LDle:} (\text{type})(\text{address})$ | Read from shadow memory, expecting data of the same type. Implemented as a dirty call. |
| Operation with one to four arguments, see Table 3 for examples. $\langle \text{op} \rangle(q_1, q_2, \ldots)$ | See Table 3. |
| Constant value. $\langle \text{literal} \rangle: (\text{type})$ or $(\text{type})\{\langle \text{literal} \rangle\}$ | Constant value zero of the same type. $\langle \text{literal} \rangle: (\text{type})$ or $(\text{type})\{0x0\}$ |
| If-then-else construct, selecting either $q_{\text{true}}$ or $q_{\text{false}}$ depending on condition. $\text{ITE(} \langle \text{condition} \rangle, q_{\text{true}}, q_{\text{false}} \rangle$ | If-then-else construct with the same condition on differentiated operands. $\text{ITE(} \langle \text{condition} \rangle, q_{\text{true}}, q_{\text{false}} \rangle$ |
| CCall to Valgrind function without side effects. | Until now, we only encountered cases without relevance to AD. |

forms differentiated expressions according to Table 2. Table 3 gives more details for the important subclass of expressions that perform an operation. From the large number of operations available in VEX, we only handle those that we consider necessary and that we suspect to be covered by our regression tests (Section 6.1). For instance, the VEX operation $\text{SinF64}$, corresponding to the x87 instruction $\text{fsin}$, is currently not handled because apparently, modern compilers and libraries do not use this instruction. For unhandled operations, Derivgrind assumes a zero derivative, and optionally outputs the containing statement for debugging purposes.

### 3.5 System Calls

Under POSIX-compliant operating systems, userspace programs accomplish (most of) their interaction with the outside by invoking functionality of the kernel. Such system calls are usually raised through software interrupts or specific instructions. Valgrind does not instrument kernel code, but enables tools to wrap system calls.

For now, Derivgrind does not use this feature. Therefore, writing data to standard, file or network streams does not export the dot values. Data read from streams into memory is endowed with the dot values previously residing in the corresponding parts of the shadow memory. In particular, Derivgrind currently cannot handle MPI multiprocessing.

### 3.6 Limitations of Valgrind

From the limitations listed in the Valgrind documentation [Seward et al. 2022], the following have a particular relevance for AD.

- Valgrind does not support 3DNow! instructions (which are anyway rarely supported by CPUs) and AVX-512 instructions (see [Volnina 2022] for current development efforts).
Table 3. VEX IR expressions performing an operation, and their forward-mode algorithmic derivatives. The placeholder \( rm \) represents an expression for the rounding mode.

| Expression \( p \)                                                                 | Differentiated expression \( \hat{p} \)                                                                 |
|--------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| Scalar floating-point arithmetic, e. g. addition of binary64s,                  | Application of the differentiation rule.                                                                  |
| AddF64(\( rm, q, s \))                                                          | AddF64(\( rm, \hat{q}, \hat{s} \))                                                                         |
| or multiplication of binary32s,                                                | AddF32(\( rm, \text{MulF32}(rm, \hat{q}, s), \text{MulF32}(rm, q, s) \))                               |
| MulF32(\( rm, q, s \))                                                         |                                                                                                            |
| SIMD floating-point arithmetic, e. g. multiplication of eight binary32s.       | Component-wise application of the differentiation rule.                                                  |
| Mul32Fx8(\( rm, q, s \))                                                       | Add32Fx8(\( rm, \text{Mul32Fx8}(rm, q, s), \text{Mul32Fx8}(rm, q, s) \))                               |
| Lowest-lane-only SIMD floating-point arithmetic, e. g. mapping the operands \( (q_0, q_1, q_2, q_3) \) and \( (s_0, s_1, s_2, s_3) \) to \( (q_0 \cdot s_0, q_1, q_2, q_3) \). | Formal application of the differentiation rule with lowest-lane-only operations, taking care to be correct outside the lowest lane also. E. g. for \( (\hat{q}_0 s_0 + q_0 \hat{s}_0, \hat{q}_1, \hat{q}_2, \hat{q}_3) \). |
| Mul32F0x4(\( q, s \))                                                          | Add32F0x4(Mul32F0x4(\( q, s \)), Mul32F0x4(\( \hat{q}, s \)))                                           |
| Floating-point conversions, e. g.                                             | Analogous application to the dot values.                                                                  |
| F64toF32(\( rm, q \))                                                          | F64toF32(\( rm, \hat{q} \))                                                                             |
| Binary reinterpretation of floating-point representations as integers and vice versa, e. g. | Analogous application to the dot values.                                                                  |
| ReinterpI64asF64(\( q \))                                                      | ReinterpI64asF64(\( \hat{q} \))                                                                         |
| SIMD (un)packing, e. g.                                                        | Analogous application to the dot values.                                                                  |
| 64x4toV256(\( q_3, q_2, q_1, q_0 \))                                           | 64x4toV256(\( \hat{q}_3, \hat{q}_2, \hat{q}_1, \hat{q}_0 \))                                            |
| Bitwise logical operations, e. g.                                             | Handling according to Section 2.4.                                                                        |
| And64(\( q, s \))                                                              | (Represented by a CCcall.)                                                                               |
| Integer arithmetic, e. g.                                                     | Not relevant for AD.                                                                                    |
| Add64(\( q, s \))                                                              | 0x0:164                                                                                                  |
| Comparisons, e. g.                                                            | Not relevant for AD.                                                                                    |
| CmpF64(\( q, s \))                                                            | 0x0:132                                                                                                  |

Valgrind encounters an unknown instruction, it sends SIGILL to the client program, triggering its termination if it did not define a signal handler.

- Valgrind replaces 80-bit by 64-bit floating-point arithmetic (as mentioned above), and performs these with partial observance of rounding modes, no support for numeric exceptions, and ignoring some SSE2 control bits. If the client program is very sensitive to floating-point errors, it might therefore behave differently when executed under Valgrind. We do not expect this to become a problem, since algorithmic derivatives of very sensitive programs are usually not meaningful.
4 ACCESSING DOT VALUES IN DERIVGRIND

Derivgrind’s instrumentation of the machine code propagates dot values alongside the execution of the compiled client program. This part of the tool, as described in Sections 2 and 3, does not rely on the source of the client program.

However, the user needs some knowledge on the internal structure of the client program in order to identify input and output variables for AD. In order to seed dot values of inputs before the propagation, and retrieve dot values of outputs afterwards, these variables must be matched to memory addresses. It is therefore natural that Derivgrind’s interfaces for setting and getting dot variables rely on the source code of the client program to some extent. We describe two interfaces in this section.

4.1 Monitor Commands Interface

Valgrind’s monitor commands mechanism enables the user to interact with Valgrind during the execution of the client program. When Valgrind is started with the command-line argument `--vgdb-error=0`, it activates its built-in gdbserver and waits for a connection, instead of executing the instrumented client program right away. The user has to connect to the gdbserver from a 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 over this connection. Derivgrind provides monitor commands to access the shadow memory, and hence the dot value of any variable. Therefore, this mechanism allows for an interactive exploration of automatic derivatives of any variable at any point of time, and with respect to any variable at any (earlier) point of time, during the execution of a client program—as long as the debugger can stop the client at these points of time, and the user can obtain memory addresses of the variables.

This condition implies that the source files which either define variables of interest, or contain lines where a breakpoint should be set, can be read by the user, and recompiled with debugging symbols (e.g. -g flag of GCC and Clang) and most optimizations turned off (e.g. -O0).

4.2 Valgrind Client Request Interface

Valgrind’s client request mechanism enables the client program to interact with the Valgrind core and tool. In contrast to the monitor commands interface (Section 4.1) where the user selects and accesses AD input and output variables interactively in a debugger, the client request interface enables the user to define them by code inserted into the client program.

To perform a request, the client program has to assemble a data structure specifying the request, load its address into a specific register, and then execute a specific sequence of machine code instructions. On a normal CPU, this instruction sequence amounts to a no-operation. When the client is running under Valgrind however, Valgrind recognizes the pattern and passes the data structure to client request handlers in the Valgrind core and tool. 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 data structure and add the specific instruction sequence using the __asm__ syntax.

Derivgrind defines and implements client requests to copy data from memory to shadow memory and vice versa. Listing 4 demonstrates the usage of the corresponding macros, which are called with a memory address, a shadow memory address, and the number of bytes to be copied. Note that when function arguments are passed by value, the AD instrumentation makes sure that their dot values are copied as well. The dot value of dotvalue is irrelevant for the setter, and unspecified for the getter.

A user of the client request interface has to insert calls to the C functions set_dotvalue (typically proceeded by an initialization of the dot value) and get_dotvalue (typically followed by an output
Listing 4. Usage of Derivgrind’s client request macros to access the shadow memory from within the client program.

```c
#include <valgrind/derivgrind.h>
double set_dotvalue(double value, double dotvalue){
    // Copy 8 bytes from the memory address &dotvalue
    // to the shadow memory address &value.
    DG_SET_DOTVALUE(&value, &dotvalue, sizeof(double));
    return value;
}
double get_dotvalue(double value){
    double dotvalue = 0.; // Return 0. if run outside Valgrind.
    // Copy 8 bytes from the shadow memory address &value
    // to the memory address &dotvalue.
    DG_GET_DOTVALUE(&value, &dotvalue, sizeof(double));
    return dotvalue;
}
```

Fig. 1. The black-box algorithmic derivative of GLIBC’s implementation of \( \sin \) and \( \log \) agrees with the analytic derivatives only within some intervals.

statement) near the lines of the client’s source code where the input and output variables are defined. It is therefore necessary that the respective parts of the source code can be edited and recompiled, and that the respective programming languages and compilers provide a “C interface”. In the special case of differentiating a Python interpreter in Section 6.3, we are able to make client requests without any modifications of the interpreter’s source code.

5 WRAPPING THE C MATH LIBRARY

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 \( \text{fsin} \) and \( \text{fcos} \), implementations of the standard library are free to perform an approximation algorithm entirely in software.
Figure 1 shows the derivatives of the GLIBC math library’s implementation of \( \sin \) and \( \log \), using the components of Derivgrind presented so far. The algorithmic derivatives match the analytic derivatives \( \cos(x) \) and \( 1/x \) only inside the intervals \([-0.126, 0.126]\) and \([0.9375, 1.0646972656 \ldots]\), respectively, and are zero outside. We further analyzed the case of \( \sin \) 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.
- 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 \( x_{\text{tab}} \) of \( 2^{-7} \) and a small remainder \( x_{\text{rem}} \). The purpose of this decomposition is to read the sine and cosine of \( x_{\text{tab}} \) from a lookup table, and to use a Taylor series for \( x_{\text{rem}} \). While the correct decomposition of \( x \) would be \( x_{\text{tab}} = 0 \) and \( x_{\text{rem}} = x \), Derivgrind erroneously computes \( x_{\text{tab}} = \hat{x} \) and \( x_{\text{rem}} = 0 \) because GLIBC performs the decomposition by adding and subtracting a big constant, relying on floating-point errors as described in Section 2.5.
- 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 \( y = x - k \cdot \frac{\pi}{2} \in [-\frac{\pi}{2}, \frac{\pi}{2}] \). As GLIBC again computes the integral factor \( k \) by a tricky exploitation of floating-point errors as described in Section 2.5, Derivgrind erroneously finds \( \hat{k} = \hat{x} \cdot \frac{2}{\pi} \) and \( \hat{y} = 0 \).

Fortunately, the Valgrind function wrapping feature allows Valgrind tools to specify functions by their names (and, if desired, the “soname” field of the containing shared object) and reroute the respective calls to wrapper functions supplied by the tool. Derivgrind wraps all C95 math functions using this mechanism. The wrappers obtain their arguments’ dot values by the client request mechanism (Section 4.2), compute the return value and its analytical derivative using the original math functions, and overwrite the return value’s dot value accordingly using the client request mechanism.

Our approach is not universal. 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. For instance, Derivgrind computes wrong derivatives in several testcases involving NumPy’s 32-bit floating-point type, because NumPy reimplements some math functions for binary32 on amd64. And the other way round, if a client program reuses \( \text{math.h} \) function names in a shared object \( \text{libm.so} \) with a different semantic or signature, it might cause unexpected behaviour.

### 6 EVALUATION

Our collection of regression tests, described in Section 6.1, checks Derivgrind’s results for a large number of small pieces of code. We apply Derivgrind to larger programs in Sections 6.2 and 6.3 for further validation, and to measure the average increase in time and memory complexity. In Section 6.2, the differentiated program is a numerical solver for Burgers’ partial differential equation (PDE). In Section 6.3, we differentiate a Python interpreter, as an example for a large pre-compiled program whose source code is not available to Derivgrind.
6.1 Regression Tests
Our test suite verifies the values and derivatives computed by Derivgrind for many combinations of
- a language and compiler: C and C++ programs are compiled by GCC and Clang, Fortran programs are compiled by GCC; Python scripts are interpreted by CPython (see more details in Section 6.3),
- a simple "algorithm" to be differentiated: elementary operations, calls to \texttt{math.h} functions, control structures, loops suitable for auto-vectorization, and OpenMP constructs;
- a floating-point type: \texttt{binary32}, \texttt{binary64}, and the 80-bit x87 double-extended precision type; and
- an architecture: x86 or amd64.

Derivgrind passes almost all of these tests, demonstrating its versatility in general. The small number of failing tests were either related to applying NumPy math functions on \texttt{binary32} arguments in CPython on amd64, or to using OpenMP constructs with GCC. The underlying issues are discussed in Sections 2.5 and 5 in more detail.

6.2 Performance Study
We apply Derivgrind to a PDE solver for the two-dimensional Burgers’ equations on a unit square, in order to differentiate a norm of the solution after the last timestep with respect to a simultaneous shift of all components of the initial state. Arithmetically, the C++ code merely involves addition, subtraction, multiplication, division, and the square root function. A related benchmark has been used in previous studies of AD performance [Blühdorn et al. 2023; Sagebaum et al. 2018, 2019, 2021]. For all the configurations considered in the following, the derivatives computed by Derivgrind match those of CoDiPack’s forward mode.

Figure 2 displays the effect of Derivgrind on the run-time. We considered $2^3 = 8$ setups, using the GCC 10.2.1 (g++) and Clang 11.0.1 (clang++) compilers, for amd64 (no flag) and x86 (-m32), with full (-O3) and without (-O0) optimization. Our time measurements refer to the difference in the system time retrieved by the client program right before and after solving the PDE. This eliminates the constant startup and finalization time of Derivgrind and the shadow memory tool, which may take up to around 2 s depending on the configuration. Averages were taken over 100 (-O3) or 10 (-O0) measurements. 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.

Each dot in Figure 2 represents a problem instance with an $n_x \times n_x$ grid and $n_t$ time steps, for $n_x = 100, 120, \ldots, 500$ and $n_t = 100, 200, \ldots, 500$. 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 amd64. 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 the forward mode of the AD tool CoDiPack, the largest slow-down factor measured by us on the setups with -O3 is approximately 3.3.

With a similar setup, Figure 3 displays the effect of Derivgrind on the required memory. Our memory measurements refer to the maximum resident set size (RSS) reported by the GNU \texttt{time} command. We consider problem instances on an $n_x \times n_x$ grid and $n_t = 4$ time steps, for $n_x = 200, 400, \ldots, 5000$, as the memory consumption hardly depends on $n_t$. As Figure 3 shows, Derivgrind doubles the memory consumption, in addition to a constant reservation of about 4.1 GB on amd64 and 20 MB on x86. On amd64, the shadow memory tool needs much more memory for its internal data structures;
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), had no significant effect on the required memory.

### 6.3 Differentiating a Python Interpreter

On many Linux systems, the default interpreter python3 for the Python programming language is CPython\(^9\). Core components of CPython are written in C. When the Python type float is used in a Python statement, CPython represents its value internally by a C double\(^10\). Basic arithmetic operations in Python are performed by CPython via the corresponding C operations\(^{11}\), and mathematical functions from the Python modules math and numpy are usually (but now always) dispatched to the C math library.

\(^{9}\)https://github.com/python/cpython
\(^{10}\)https://github.com/python/cpython/blob/787498cbb7d1c7115a7af4435efb7f607b10ed1/Include/cpython/floatobject.h#L7
\(^{11}\)https://github.com/python/cpython/blob/787498cbb7d1c7115a7af4435efb7f607b10ed1/Objects/floatobject.c#L597
Listing 5. Usage of the Python extension module derivgrind to access the shadow memory from within a Python script, assuming the interpreter is running under Valgrind.

```python
import derivgrind
x = derivgrind.set_dotvalue(4,1)
y = x**x
print(derivgrind.get_dotvalue(y))
```

Derivgrind can thus be used to differentiate a Python script, by applying it to CPython interpreting the Python script, i.e.,

```
valgrind --tool=derivgrind python3 ⟨Python script⟩ ⟨arguments for Python script⟩
```

In our test system based on Debian 11.6 on amd64, we obtain python3 as a pre-built package of CPython 3.9.2 from a software repository, so the source code of CPython is not available on the system. In order to access dot values of Python variables, we use pybind11 [Jakob et al. 2017] to create a Python extension module derivgrind, i.e., a shared object compliant with the Python/C API. At run-time, when CPython reads the Python statement `import derivgrind`, it dynamically loads derivgrind.so. When functions of this module are used on the Python side as shown in Listing 5, CPython calls the corresponding functions of Listing 4, which were compiled into the shared object. In short, insertions into the source code of CPython are not necessary because CPython exposes Python variables to user-supplied C code at runtime.

We reimplemented the Burgers benchmark as a Python script, using standard Python lists instead of C arrays, `math.sqrt` to compute the square root, and our extension module derivgrind instead of direct client request macros. We use the setup of Section 6.2 with a 200 × 200 grid and 200 time steps. The computed values and derivatives match those of the C++ program. Time measurements, taken via clock reads from the Python program, result in 9.5 s for CPython running natively and 650 s for CPython running under Valgrind, corresponding to a slow-down factor of 68.

7 SUMMARY

With the new AD tool Derivgrind, we have demonstrated a methodology to augment compiled programs with forward-mode AD logic, independent of their source code. Derivgrind handles x86 and amd64 machine code through the VEX intermediate representation of the Valgrind framework. Every temporary, register and memory byte is shadowed to keep track of the respective dot values. For statements with floating-point expressions or copy operations, Derivgrind updates the shadows according to the respective analytical rules of differentiation.

Real arithmetic can also be performed by integer or logic operations, in manifold ways. Our current lack of a systematic approach to detect all the real arithmetic “hidden” in a portion of machine code would be a fundamental obstacle if we sought 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. To this end, we have set up an extensive test suite, which checks various simple programs produced by the GCC and Clang compilers, as well as the precompiled CPython interpreter as it runs various Python scripts. The results indicate that unless explicitly instructed otherwise by the source code, actual compilers only very rarely realize unsupported bit-tricks.

In order to identify the input and output variables of the differentiation task, and to set or get the respective dot values, generally the parts of the client’s source code containing their definitions
must be accessible, in one of the following two ways. For the monitor commands interface, these parts of the source must be compiled with debugging symbols and optimization turned off. For the client request interface, they must be augmented by calls to C functions, and recompiled. If the client program exposes the variables of interest via a suitable API, this offers another way to access their dot values without any modifications of the client’s source code. We used this approach to demonstrate AD for Python programs, by applying Derivgrind to the Python interpreter and injecting additional C code via Python extension modules.

Time measurements on various client executables found that Derivgrind scales their run-time by a factor between 30 and 75, in addition to a start-up time of a few seconds. While this is considerably slower than existing AD tools tuned for high performance, Derivgrind is applicable for a much wider range of software, with less integration efforts. The real arithmetic between input and output variables is provided to Derivgrind as machine code only, so it does not matter whether it has been compiled from a variety of programming languages, uses pre-compiled libraries, or comes in the shape of an interpreter running a script. Seeding inputs and retrieving output derivatives can be as easy as stopping the client in the debugger and issuing monitor commands there.

ACKNOWLEDGMENTS

Max Aehle gratefully acknowledges funding from the research training group SIVERT by the German federal state of Rhineland-Palatinate.

We are grateful to the authors of Valgrind for creating such a highly versatile framework, and to Karl Cronburg for sharing his shadow memory library\textsuperscript{12} that was very helpful at an early development stage of Derivgrind.

REFERENCES

AMD64 Technology. 2021. *AMD64 Architecture Programmer’s Manual, Volumes 1–5*. Technical Report. Revision 4.04.
Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic Differentiation in Machine Learning: A Survey. *J. Mach. Learn. Res.* 18, 1 (2017), 5595–5637.
Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-Point Accuracy Problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) (PLDI ’12). Association for Computing Machinery, New York, NY, USA, 453–462. https://doi.org/10.1145/2254064.2254118
Johannes Blühdorn, Max Sagebaum, and Nicolas R. Gauger. 2023. Event-Based Automatic Differentiation of OpenMP with OpDiLib. *ACM Trans. Math. Softw.* 49, 1, Article 3 (mar 2023), 31 pages. https://doi.org/10.1145/3570159
Will Drewry and Tavis Ormandy. 2007. Flayer: Exposing Application Internals. In *First USENIX Workshop on Offensive Technologies (WOOT 07)* (Boston, MA, USA).USENIX Association, Berkeley, CA, USA, 9 pages. https://www.usenix.org/conference/woot-07/flayer-exposing-application-internals
Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2 (June 2007), 13–es. https://doi.org/10.1145/1236463.1236468
François Févotte and Bruno Lathuilière. 2017. Studying the Numerical Quality of an Industrial Computing Code: A Case Study on code_aster. In *10th International Workshop on Numerical Software Verification (NSV)* (Heidelberg, Germany), Alessandro Abate and Sylvie Boldo (Eds.). Springer International Publishing AG, Cham, Switzerland, 61–80. https://doi.org/10.1007/978-3-319-63501-9_5
François Févotte and Bruno Lathuilière. 2019. Debugging and Optimization of HPC Programs with the Verrou Tool. In *International Workshop on Software Correctness for HPC Applications (Correctness)* (Denver, CO, USA). Curran Associates, Inc., New York, NY, USA, 1–10. https://doi.org/10.1109/Correctness49594.2019.00006
Hadjrien Grasland, François Févotte, Bruno Lathuilière, and David Chamont. 2019. Floating-point profiling of ACTS using Verrou. *EPJ Web Conf.* 214 (2019), 05025. https://doi.org/10.1051/epjconf/201921405025
Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. https://doi.org/10.1137/1.9780898717761

\textsuperscript{12}https://github.com/cronburg/shadow-memory
