Compiler Support for Operator Overloading and Algorithmic Differentiation in C++

Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation von Alexander Hück, M.Sc. aus Wiesbaden
Tag der Einreichung: 16.01.2020, Tag der Prüfung: 28.02.2020

1. Gutachten: Prof. Dr. Christian Bischof
2. Gutachten: Prof. Dr. Martin Bücker
Darmstadt – D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
Scientific Computing

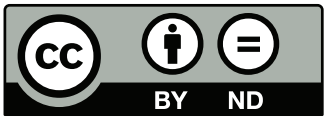
Compiler Support for Operator Overloading and Algorithmic Differentiation in C++

Doctoral thesis by Alexander Hück, M.Sc.

1. Review: Prof. Dr. Christian Bischof
2. Review: Prof. Dr. Martin Bücker

Date of submission: 16.01.2020
Date of thesis defense: 28.02.2020

Darmstadt – D 17



Erklärungen laut Promotionsordnung

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 16.01.2020

Alexander Hück

Abstract

Multiphysics software needs derivatives for, e.g., solving a system of non-linear equations, conducting model verification, or sensitivity studies. In C++, algorithmic differentiation (AD), based on operator overloading (overloading), can be used to calculate derivatives up to machine precision. To that end, the built-in floating-point type is replaced by the user-defined AD type. It overloads all required operators, and calculates the original value and the corresponding derivative based on the chain rule of calculus. While changing the underlying type seems straightforward, several complications arise concerning software and performance engineering. This includes (1) fundamental language restrictions of C++ w.r.t. user-defined types, (2) type correctness of distributed computations with the Message Passing Interface (MPI) library, and (3) identification and mitigation of AD induced overheads. To handle these issues, AD experts may spend a significant amount of time to enhance a code with AD, verify the derivatives and ensure optimal application performance. Hence, in this thesis, we propose a modern compiler-based tooling approach to support and accelerate the AD-enhancement process of C++ target codes. In particular, we make contributions to three aspects of AD.

The initial type change While the change to the AD type in a target code is conceptually straightforward, the type change often leads to a multitude of compiler error messages. This is due to the different treatment of built-in floating-point types and user-defined types by the C++ language standard. Previously legal code constructs in the target code subsequently violate the language standard when the built-in floating-point type is replaced with a user-defined AD type. We identify and classify these problematic code constructs and their root cause is shown. Solutions by localized source transformation are proposed. To automate this rather mechanical process, we develop a static code analyser and source transformation tool, called OO-Lint, based on the Clang compiler framework. It flags instances of these problematic code constructs and applies source transformations to make the code compliant with the requirements of the language standard. To show the

overall relevance of complications with user-defined types, OO-Lint is applied to several well-known scientific codes, some of which have already been AD enhanced by others. In all of these applications, except the ones manually treated for AD overloading, problematic code constructs are detected.

Type correctness of MPI communication MPI is the de-facto standard for programming high performance, distributed applications. At the same time, MPI has a complex interface whose usage can be error-prone. For instance, MPI derived data types require manual construction by specifying memory locations of the underlying data. Specifying wrong offsets can lead to subtle bugs that are hard to detect. In the context of AD, special libraries exist that handle the required derivative book-keeping by replacing the MPI communication calls with overloaded variants. However, on top of the AD type change, the MPI communication routines have to be changed manually. In addition, the AD type fundamentally changes memory layout assumptions as it has a different extent than the built-in types. Previously legal layout assumptions have, thus, to be reverified. As a remedy, to detect any type-related errors, we developed a memory sanitizer tool, called TypeART, based on the LLVM compiler framework and the MPI correctness checker MUST. It tracks all memory allocations relevant to MPI communication to allow for checking the underlying type and extent of the typeless memory buffer address passed to any MPI routine. The overhead induced by TypeART w.r.t. several target applications is manageable.

AD domain-specific profiling Applying AD in a black-box manner, without consideration of the target code structure, can have a significant impact on both runtime and memory consumption. An AD expert is usually required to apply further AD-related optimizations for the reduction of these induced overheads. Traditional profiling techniques are, however, insufficient as they do not reveal any AD domain-specific metrics. Of interest for AD code optimization are, e.g., specific code patterns, especially on a function level, that can be treated efficiently with AD. To that end, we developed a static profiling tool, called ProAD, based on the LLVM compiler framework. For each function, it generates the computational graph based on the static data flow of the floating-point variables. The framework supports pattern analysis on the computational graph to identify the optimal application of the chain rule. We show the potential of the optimal application of AD with two case studies. In both cases, significant runtime improvements can be achieved when the knowledge of the code structure, provided by our tool, is exploited. For instance, with a stencil code, a speedup factor of about 13 is achieved compared to a naive application of AD and a factor of 1.2 compared to hand-written derivative code.

Zusammenfassung

Multiphysik-Software benötigt Ableitungen, um beispielsweise ein System nichtlinearer Gleichungen zu lösen, oder Modellverifizierungen und Sensitivitätsstudien durchzuführen. In C++ kann die Algorithmische Differenzierung (AD) basierend auf Operator Overloading (Überladung) verwendet werden, um Ableitungen bis zur Maschinengenauigkeit zu berechnen. Zu diesem Zweck wird der integrierte Gleitkommatyp durch den benutzerdefinierten AD-Typ ersetzt. Er überlädt alle erforderlichen Operatoren, und berechnet den ursprünglichen Wert und die entsprechende Ableitung basierend auf der Kettenregel der Differentialrechnung. Während das Ändern des zugrunde liegenden Typs unkompliziert erscheint, treten als Konsequenz einige Komplikationen in Bezug auf Software und Performance Engineering auf. Dies beinhaltet (1) grundlegende Spracheinschränkungen von C++ bezüglich benutzerdefinierter Typen, (2) Typkorrektheit verteilter Berechnungen mit dem Message Passing Interface (MPI) und (3) Identifikation und Behandlung von AD-induziertem Overhead. Um diese Komplikationen zu bewältigen, können AD-Experten viel Zeit darauf verwenden, einen Code mit AD zu versehen, die Ableitungen zu überprüfen und eine optimale Anwendungsleistung sicherzustellen. Daher schlagen wir in dieser Arbeit einen modernen compilerbasierten Tooling-Ansatz vor, um den AD-Verbesserungsprozess von C++-Zielcodes zu unterstützen und zu beschleunigen. Insbesondere leisten wir Beiträge zu drei Aspekten des Differenzierens mit AD.

Die initiale Typänderung Während die Änderung des AD-Typs in einem Zielcode konzeptionell unkompliziert ist, führt die Typänderung häufig zu einer Vielzahl von Compilerfehlermeldungen. Dies ist auf die unterschiedliche Behandlung der integrierten Gleitkommatypen und der benutzerdefinierten Typen durch den C++-Sprachstandard zurückzuführen. Zuvor gültige Codekonstruktionen im Zielcode verletzen anschließend den Sprachstandard, wenn der integrierte Gleitkommatyp durch einen benutzerdefinierten AD-Typ ersetzt wird. Wir identifizieren und klassifizieren diese problematischen Codekonstrukte und zeigen ihre Ursache auf. Zu jedem identifizierten Codekonstrukt werden

Lösungen durch lokalisierte Quelltexttransformation vorgeschlagen. Um diesen eher mechanischen Prozess zu automatisieren, entwickeln wir auf der Grundlage des Clang-Compiler-Frameworks ein statisches Code-Analyse- und Quelltexttransformationswerkzeug namens OO-Lint. Es kennzeichnet Instanzen dieser problematischen Codekonstrukte und wendet Quelltexttransformationen an, um sicherzustellen, dass der Code nach der AD-Typänderung den Anforderungen des Sprachstandards entspricht. Um die allgemeine Relevanz von Komplikationen mit benutzerdefinierten Typen aufzuzeigen, wird OO-Lint auf mehrere bekannte wissenschaftliche Codes angewendet, von denen einige bereits durch andere AD-erweitert wurden. In all diesen Anwendungen, mit Ausnahme derer, die manuell für die AD-Überladung behandelt wurden, werden problematische Codekonstrukte erkannt.

Datentyp-Korrektheit von MPI-Kommunikation MPI ist der faktische Standard für die Programmierung von verteilten Hochleistungsanwendungen. Gleichzeitig verfügt es über eine komplexe Programmierschnittstelle, deren Verwendung fehleranfällig sein kann. Beispielsweise erfordern von MPI abgeleitete Datentypen eine manuelle Erstellung, indem Speicherorte der zugrunde liegenden Daten angegeben werden. Die Angabe falscher Speicheradressen kann zu subtilen Fehlern führen, die nur schwer zu erkennen sind. Im Kontext von AD existieren spezielle Bibliotheken, die die erforderliche Buchhaltung für Ableitungen verwalten, indem sie die MPI-Kommunikationsaufrufe durch überladene Varianten ersetzen. Zusätzlich zur Änderung des AD-Typs müssen die MPI-Kommunikationsroutinen jedoch manuell geändert werden. Darüber hinaus werden durch den AD-Typ die Annahmen zur Speicherauslegung grundlegend geändert, da er ein anderes Ausmaß aufweist als die integrierten Typen. Bisherige Annahmen zur Speicherauslegung müssen daher erneut überprüft werden. Um typbezogene Fehler zu erkennen, haben wir das auf dem LLVM-Compiler-Framework und dem MPI-Korrektheitsprüfer MUST basierende Tool TypeART entwickelt. Es verfolgt alle für die MPI-Kommunikation relevanten Speicherallokationen, um den zugrunde liegenden Typ und den Umfang der typlosen Speicherpufferadresse zu überprüfen, die an eine MPI-Routine übergeben wird. Das Anwenden von TypeART auf mehrere Zielanwendungen zeigt, dass der verursachte Overhead akzeptabel ist.

AD-domänenspezifische Profilerstellung Das Black-Box-Anwenden von AD ohne Berücksichtigung der Zielcodestruktur kann sich erheblich auf die Laufzeit und den Speicherverbrauch auswirken. Ein AD-Experte muss in der Regel weitere AD-bezogene Optimierungen vornehmen, um diese induzierten Berechnungskosten zu reduzieren. Herkömm-

liche Profiling-Techniken sind jedoch unzureichend, da sie keine domänenspezifischen AD-Metriken anzeigen. Von Interesse für die AD-Code-Optimierung sind beispielsweise bestimmte Codemuster, insbesondere auf Funktionsebene, die mit AD effizient behandelt werden können. Zu diesem Zweck haben wir ein statisches Profiling-Tool namens ProAD entwickelt, das auf dem LLVM-Compiler-Framework basiert. Für jede Funktion wird der Berechnungsgraph basierend auf dem statischen Datenfluss der Gleitkommavariablen generiert. Das Framework unterstützt die Musteranalyse im Berechnungsgraph, um die optimale Anwendung der Kettenregel zu ermitteln. Das Potenzial der optimalen Anwendung von AD zeigen wir anhand von zwei Fallstudien. In beiden Fällen können erhebliche Laufzeitverbesserungen erzielt werden, wenn die von unserem Tool bereitgestellten Kenntnisse über die Codestruktur genutzt werden. Beispielsweise wird bei einem Stencil-Code ein Beschleunigungsfaktor von ungefähr 13 im Vergleich zu einer naiven Anwendung von AD und ein Faktor von $1{,}2$ im Vergleich zu einem handgeschriebenen abgeleiteten Code erreicht.

Contents

1 Introduction

Computational science is a major contributor to science and engineering progress, especially in domains where experimentation and validation can be difficult. At the same time, modeling fidelity has consistently risen driven by algorithmic advances and greater capabilities in High-Performance Computing (HPC).

HPC software complexity: Time to solution is emphasized The rising system complexity, however, makes the process of writing and maintaining a multiphysics code for modern HPC clusters time-consuming. Aspects which impact the complexity and workload of writing HPC simulation codes encompass the complete software (development) life cycle: (1) Writing the software with *maintainability* as a concern. (2) *Validating and ensuring correctness* of the software results. Correctness (trustworthiness) in the numerical output is an "overriding software quality" as stated in [75]. (3) Ensuring *efficiency and scalability* of the scientific code.

Until a simulation code has reached a stage of maturity and approval regarding the aforementioned software properties, years of development time are usually invested by computational scientists and other experts. These codes can, thus, be long-lived, with a multi-decade project life cycle, until the software development is stopped and a successor code is developed [113]. The cost of software (monetary factors or learning curve) is a relevant factor [57]. Unsurprisingly, the software engineering lifecycle and economics in the HPC context is different [11]: Execution time is not the sole software engineering emphasis but *time to solution* — development and execution time — must be the focus.

A common need in multiphysics applications: Derivatives Multiphysics applications typically need derivatives, e.g., to solve a system of non-linear equations. Likewise, derivatives are required for validating the underlying mathematical models (e.g., [5; 42]),

with sensitivity studies [21] or data assimilation [6]. The provision of derivatives has, thus, been identified as a critical component of multiphysics codes [76].

Derivatives are implemented with Finite Differences (FD) by perturbing the model input repeatedly, but this comes at a loss of precision [74; 104]. Symbolic or hand-derived codes, if possible, are hard to produce and maintain due to the large code size. Any code change also may require extensive maintenance work.

In contrast, Algorithmic Differentiation (AD) (sometimes called *automatic* differentiation, [45; 104]) allows for the computation of derivatives up to machine precision. AD is based on the principle of the application of the chain rule of calculus [17]. A code consists of a long chain of arithmetic operations and functions of which the analytical derivatives are known a priori. Applying the chain rule, therefore, allows for the propagation of derivative values through the code.

The application of AD can be done by source transformation, Operator Overloading (overloading) or a combination thereof. The source transformation approach modifies the original, or *primal*, code augmenting it with derivative statements. However, the source transformation AD tools, as of now, can not handle all the complexities of a C++ code base, which is the focus of this thesis. Overloading is, therefore, the only feasible way for complex C++ codes.

Applying an AD overloading tool starts with the replacement of the underlying floating-point type with the new user-defined AD type. The new type, typically, completely encapsulates [32] the primal value, while also carrying the derivative information in an implementation-dependent manner. In addition, it overloads all required arithmetical operators and mathematical functions for the computation of derivatives.

AD does not equal automatic: Augmentation and performance concerns To apply AD, a developer starts with (1) the introduction of the AD type, (2) the integration of any underlying libraries into the AD-related computation, and, finally (3) the initialization of the AD variables (called *seeding*) to compute and then extract the derivatives [108]. In the so-called black box AD approach, the AD type replaces the floating-point type globally and no distinction between actives, i.e., variables with an influence on the derivative, and passives, i.e., variables whose derivative are zero, w.r.t. AD is made [45; 117; 130]. The advantage, in particular, is that the derivative computation is always in sync when additional code to gain new functionality is added to a project.

The perceived simplicity of AD was the reason for it being "automatic". However, it brings several complications concerning software and performance engineering. Among others, sources for complications are (1) fundamental restrictions of the programming language due to the treatment of user-defined types, (2) complex underlying libraries which need to be integrated into the AD computation, (3) Message Passing Interface (MPI) [98] communication, (4) efficiency regarding AD-induced overheads, and, finally (5) the verification of the correctness of the derivatives. These aspects require detailed knowledge of the target code and the AD tool's features and limitations. As there is no universally valid approach to applying AD optimally w.r.t. performance and correctness, each target code poses different challenges to the AD experts:

Language restrictions with overloading. The C++ language standard treats built-in floating-point types differently compared to user-defined types, i.e., the AD overloading type. This can lead to compile-time errors originating from code locations that were legal before the switch to the AD overloading type, independent of the targeted C++ language revision [65–69]. For instance, floating-point value types in branch conditions are automatically converted to bool values by the compiler. In the case of user-defined types, the compiler can only convert it if the correct conversion operation is explicitly provided by the AD tool. These and other implicit conversions happen many times in codes, often without the developer's knowledge. As such, unexpected errors can occur after a type change.

Underlying libraries. External libraries and other language constructs sometimes necessitate special handling with AD, due to (1) library codes or interfaces which can not be made compatible with the AD type (e.g., C library functions), (2) libraries that would result in too high of a workload to differentiate them, or (3) the additional computational overhead of the differentiated library requires care.

If a library is not relevant for the derivative computation, say, C library functions for printing to the console, they are typically wrapped to call them with the primal value only. On the other hand, libraries relevant to AD are integrated as a so called *external* function into the overall computation. Unless a differentiated version of the library is available, the developer has to provide the derivatives of that external code section manually to the AD tool, as the library itself is, again, only called with the primal values. The external function concept is also relevant for reducing the AD induced overheads, as discussed further below.

MPI communication. MPI is the de facto standard for large scale distributed applications in HPC (e.g., [23; 78]). AD tools use special MPI libraries [119; 121; 131], which

handle the required derivative book-keeping before invoking the standard MPI routine. However, these libraries (1) support a different set of AD tools, and (2) they may not support all MPI communication routines. In addition, the MPI routines relevant to the derivative computation have to be changed manually on top of the type change to the AD tool. Hence, two significant changes happen to the code base which require care. Any type layout assumptions, or pointer arithmetic w.r.t. the typeless MPI buffers must therefore be verified (again), else subtle memory access errors may occur.

Efficiency of AD. AD-enhanced codes can be quickly limited by, e.g., memory constraints of the hardware. To achieve the most efficient computation of derivatives w.r.t. time and memory, a hierarchical view of the code is required [14]. The associativity of the chain rule allows for logically dividing the source code in regions with differing levels of abstraction. Applying the chain rule to arithmetic operations on scalars individually is the lowest level of the hierarchy — a level on which overloading in C++ applies. However, this simplistic view is not runtime and memory optimal as each operation returns temporary AD objects. Assignment statements, hierarchically higher than expressions, on the other hand, are handled as a whole by most modern AD tools using sophisticated template techniques [134]. With this technique, temporaries are avoided compared to evaluating each expression of the statement individually, and an optimizing compiler can generate efficient code for these.

Entire procedures can also be handled on a higher level of abstraction: Solving a linear system of equations, for example, can be integrated in the AD computation on a matrix and vector level differentiation [39; 105]. It describes the handling of code constructs, i.e., the linear system solver, symbolically while other code parts are computed with the AD type. The solver library is then treated as a black box in the AD context: The invocation happens according to the solver API, and the AD expert is responsible for the manual integration of the symbolic derivative into the AD tool at the appropriate point of the computation. This approach significantly reduces memory requirements of the AD code [130].

Verification of derivatives. The verification of derivatives is part of the general concept of software correctness [114]. AD tools typically have a test base (of unit tests) to ensure correct operation. However, for each target code, testing of the resulting derivatives has to be done by (1) an expert validating the plausibility of the results for, e.g., sensitivity studies, (2) validation against FD, or, ideally, (3) test cases where symbolic or hand-derived solutions exist. This is not always straight forward if, say,

the convergence behaviour of an iterative solver of the simulation code influences the accuracy of the AD derivatives [2].

Time to solution: A tooling perspective The aforementioned aspects and complexities of *efficient* derivatives with AD usually require an iterative development process for AD-enhancement of a code. Initial development of AD capabilities, i.e., dealing with language restrictions and underlying libraries, is followed by an iterative process of (AD) performance improvements, with validation and verification as the major focus of all phases. Depending on the code, the impact of AD on efficiency can be severe, with unexpected slow computational performance or simulations simply running out of memory. Detailed knowledge of the code is required to identify hot spots of AD that consume most of the computational resources (e.g., memory consumption) and to handle them.

This development cycle can take years until completion. For instance, the AD augmentation of the Ice Sheet System Model (ISSM) [83] took approximately four years of development time until it was considered production ready [84]. However, a tight project schedule can lead to a funding cut if an important milestone is missed [113], and, also, allocated development resources which could instead be spent on performance tuning [122] or integrating new algorithms and models are very relevant aspects in HPC software engineering. Unsurprisingly then, a need for *better* tools has been identified in the field of HPC, including, among others, testing tools, performance analysis tools and parallel debuggers [113].

A tooling approach In this thesis, on the basis of our experiences and the works of others dealing with AD codes, we propose modern compiler-based tooling to support the process of enhancing a code with AD-based derivative computations. In particular, we make the following contributions:

1. *The type change*: Due to the language restrictions for user-defined types, during the AD augmentation, the developer is forced to (1) iteratively compile the code, (2) identify the root cause of the error, and, finally, (3) fix it. Our experiences of this tedious work lead to an extensive analysis. We classify these problematic code constructs and propose localized code changes to remedy these problems — offering a blue print to other developers for ensuring compatibility with AD. We subsequently developed a static code analyser which detects and reports these problematic code locations and, also, applies the proposed code changes automatically.

2. *Correctness*: When changing (1) the underlying floating-point type, and (2) all MPI invocations in a code base, a complete reverification of the (changed) type-related operations is necessary. This includes, e.g., the registration of MPI user-defined types which are registered by the developer by providing memory address offsets. Such a verification process can be tedious in large code bases. To that end, we developed a tool to track the type and extent of all memory allocations in a program during runtime. It acts as an extension to the MPI correctness checker MUST [54]. This allows for, e.g., ensuring the correctness of the data layout expected by the MPI library compared to the actual allocation layout in main memory.
3. *Profiling for AD*: For an efficient AD use, detailed insights of the target application are required. Performance engineers typically generate these insights with profiling tools followed by manual code inspection. Traditional profiling tools, however, do not have AD aware semantics to help gain necessary insights. To that end, an AD domain-specific profiling toolchain is needed. We present such a framework. It collects, for each function, AD-related code characteristics statically. This allows for reasoning about the optimal application of AD by, e.g., analysing the data flow in functions.

Overview Chapter 2 presents fundamental and advanced techniques of AD. Chapter 3 discusses, based on two case studies, aspects of (1) introducing the AD overloading type to a code base, and (2) exchanging AD overloading tools in a code base that has a verified AD version. The required (code) changes are juxtaposed with codes of similar complexity that have been augmented with AD by other experts. The approach to find and transform problematic code constructs w.r.t. the AD type change is presented in Chapter 4. Chapter 5 covers MPI-related type correctness facilitated by memory allocation type tracking using static compiler instrumentation and a runtime type tracking library. AD domain-specific profiling of applications is discussed in Chapter 6. The presented profiling tool statically collects metrics of functions in the target code for enabling AD performance modeling. Finally, Chapter 7 concludes this thesis and gives an outlook on where AD can further benefit from compiler tooling. Common abbreviations for this thesis are defined in Section L.1.

2 Algorithmic Differentiation

AD [45] denotes the semantic augmentation of codes in order to compute derivatives. In the context of AD, any code is assumed to be a sequence of mathematical operations (e.g., `+`) and functions (e.g., `sin`) which have known analytical derivatives. Thus, the chain rule of differentiation can be applied to each statement, resulting in the propagation of derivatives through the code.

The semantic augmentation for AD is either done by a source transformation, e.g., [49; 102; 132; 136], by overloading, e.g., ADOL-C [46; 137], Adept [56], dco [90] and CoDiPack [117; 118], or a combination thereof, e.g., ADiMat [13]. The listed overloading tools mostly target CPU-based computations, however, some implementations target GPUs [71; 88].

Source transformation takes a program P, augments it with derivative statements, resulting in a new program $\widetilde{P}$. Overloading is based on redeclaring the built-in floating-point type T to a user-defined AD type $\widetilde{T}$. It stores the original value of T, called *primal value*, and overloads all required operations to perform additional derivative computations. Many AD tools based on overloading and source transformation can be found on the AD community website (`www.autodiff.org`).

This chapter is structured as follows. After introducing the notation, the fundamental concepts of AD are introduced in Section 2.1. The software implementation of AD is discussed in Section 2.2. In particular, the focus is on the overloading concept which is typically applied to C++ codes. Finally, advanced techniques of AD are briefly described in Section 2.3. Here, the concerns are the reduction of runtime and memory overheads induced by AD.

Notation The following mathematical notation is used for the introduction to AD. Bold letters are vectors $\mathbf{v}$, upper case letters are matrices M (if not otherwise specified), and Φ is used to represent elemental functions, e.g., a multiplication. An element-wise assignment

of three values, say, has the form $x_{1\ldots3} = y_{1\ldots3}$. The notation for derivative values and related definitions is based on [45]. Another introduction to AD, with a slightly different notation, is [104].

2.1 Fundamentals

Given a continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}^m$ with $\mathbf{y} = f(\mathbf{x})$, the Jacobian J_f is defined as

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_k} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial f_j}{\partial x_1} & \cdots & \frac{\partial f_j}{\partial x_k} & \cdots & \frac{\partial f_j}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_k} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{2.1}$$

and the derivatives of f with $m = 1$ is therefore a row vector, i.e., called a transposed gradient,

$$\frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n \, . \tag{2.2}$$

Two main AD modes exist, the Forward Mode (FM) and the Reverse Mode (RM). They differ in the way the chain rule is applied to the function. The assumption is that each computer program is a composite function $f(\mathbf{x}) \mapsto (g \circ h)(\mathbf{x}) = g(h(\mathbf{x}))$ and applying the chain rule yields $\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial g}{\partial h}\frac{\partial h}{\partial \mathbf{x}} \equiv J_{g\circ h} = J_g(h(\mathbf{x})) \cdot J_h(\mathbf{x})$.

More generalized, based on a composition of K functions,

$$f = f^K \circ f^{K-1} \circ \ldots \circ f^1 \tag{2.3}$$

the application of the chain rule to (2.3) yields

$$J_f = J_f^K \cdot J_f^{K-1} \cdot \ldots \cdot J_f^1 \, . \tag{2.4}$$

The application of the chain rule differs with the AD modes, however, they both compute derivative values up to machine precision.

Single assignment code AD is based on the fact, that each computer program can be interpreted as a sequential chain of many elemental operations. In the context of C++ codes, these include binary or unary operations and the common mathematical functions part of the standard library. This Single Assignment Code (SAC) decomposition of AD target codes comes naturally with the overloading approach, where, e.g., each binary operation is overloaded and, thus, calculates and returns the derivative value for this single operation.

The program is decomposed into a call sequence of the elemental functions Φ_i, which maps the n *independent* inputs to the m *dependent* outputs, resulting in the creation of p intermediate values. The mappings are therefore

$$\begin{aligned} &\text{for } i = 1, \ldots, n \\ &\quad v_i = x_i && \text{(2.5a)} \\ &\text{for } i = n+1, \ldots, n+p \\ &\quad v_i = \Phi_i(v_j)_{j \prec i} && \text{(2.5b)} \\ &\text{for } i = 1, \ldots, m \\ &\quad y_i = v_{n+p+1-i} && \text{(2.5c)} \end{aligned}$$

where (2.5a) is the initialization for the independents, (2.5b) is the mapping to the intermediates with $j \prec i$ representing a direct dependence of the value v_i on v_j, and the dependents are assigned in (2.5c).

2.1.1 Forward mode

The FM computes directional derivatives of the form

$$\dot{\mathbf{y}} = \frac{\partial f}{\partial \mathbf{x}} \dot{\mathbf{x}} = J_f \cdot \dot{\mathbf{x}} \tag{2.6}$$

with $\dot{\mathbf{x}} \in \mathbb{R}^n$ being the direction of the input vectors in which the directional derivative is evaluated. $\dot{\mathbf{y}} \in \mathbb{R}^m$ is the directional derivative of f in the direction $\dot{\mathbf{x}}$. Based on (2.5b), applying (2.6) to every intermediate value of the elemental operations of f, propagates

the derivatives along the program flow

$$\begin{aligned} &\text{for } i = n+1, \ldots, n+p: \\ &\quad \dot{v}_i = \sum_{j \prec i} \frac{\partial \Phi_i(v_j)}{\partial v_j} \dot{v}_j \\ &\quad v_i = \Phi_i(v_j)_{j \prec i} \end{aligned} \tag{2.7}$$

with initial values of the independents, see (2.5a), $v_{1\ldots n} = x_{1\ldots n}$ and the corresponding derivative directions $\dot{v}_{1\ldots n} = \dot{x}_{1\ldots n}$. The assignment of the dependent values is done analogously, see (2.5c).

Seeding Initializing the directional derivative $\dot{\mathbf{x}}$ is called *seeding*. The Jacobian is computed column-wise by setting $\dot{\mathbf{x}}$ to each of the n Cartesian basis vectors $\mathbf{e}_i \in \mathbb{R}^n$ and evaluating the program for each vector. To compute the full Jacobian of f, FM has, thus, a time complexity of $\mathcal{O}(n) \cdot t_f$ with t_f being the time it takes for one evaluation of f.

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_k} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial f_j}{\partial x_1} & \cdots & \frac{\partial f_j}{\partial x_k} & \cdots & \frac{\partial f_j}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_k} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

After k function evaluations of f with the FM

To illustrate the FM with (2.7) and the corresponding seeding to generate the full Jacobian, the function $f : \mathbb{R}^2 \to \mathbb{R}$ and its Jacobian are defined as

$$f(x_1, x_2) = x_1 x_2 \sin(x_1) \tag{2.8}$$

$$J_f = [x_2(\sin(x_1) + x_1 \cos(x_1)),\ x_1 \sin(x_1)] \in \mathbb{R}^2\ . \tag{2.9}$$

Interpreting (2.8) as a SAC, and applying (2.7) with $n = 2$, $m = 1$ and $p = 3$ results in

$$\begin{aligned}
\dot{v}_{1\ldots2} &= \dot{x}_{1\ldots2} \\
v_{1\ldots2} &= x_{1\ldots2} \\
\dot{v}_3 &= \dot{v}_1 v_2 + v_1 \dot{v}_2 \\
v_3 &= \Phi_3 = v_1 v_2 \\
\dot{v}_4 &= \cos(v_1)\dot{v}_1 \\
v_4 &= \Phi_4 = \sin(v_1) \\
\dot{v}_5 &= \dot{v}_3 v_4 + v_3 \dot{v}_4 \\
v_5 &= \Phi_5 = v_3 v_4 \\
\dot{y}, y &= \dot{v}_5, v_5
\end{aligned} \tag{2.10}$$

where each variable has a corresponding derivative value, see (2.7).

The SAC forms the derivative equation $J_f \cdot [\dot{x}_1, \dot{x}_2]^T$, see (2.9). Setting the initial values $\dot{x}_1 = 0, \dot{x}_2 = 1$, the second column of the Jacobian is computed, $J_f \cdot [0, 1]^T = x_1 \sin(x_1)$. Repeating the execution a second time, produces the full Jacobian. The process of (partially) assembling the Jacobian from the derivative values $\dot{y}$ is sometimes called *extracting* or *harvesting*, see Section 2.2.3.

The memory requirement of the FM is approximately twice that of the primal program. For each evaluation of the target function f, temporarily, each variable has a corresponding additional partial derivative along the program flow until the scope of the function is left, and memory can be freed. It follows that the memory is thus $2 \cdot \mathcal{O}(m_f)$ with m_f being the maximum memory requirement of the function f during evaluation.

Vector mode extension The FM, as described, requires n evaluations of the target function f for the full Jacobian. The additional overhead of repeating the primal computation motivates two different approaches. The *pure mode* restricts the computation to derivative statements and, thus, omits (repeated) calculations of primals [37; 136]. The so-called *vector mode* of the FM is employed more commonly. Instead of each $\dot{x}_i$ carrying only a single seed value, as in the example above, the derivative values are extended to vectors $\dot{\mathbf{x}}_i \in \mathbb{R}^n$. This results in the general product $J_f S$ where $S \in \mathbb{R}^{n \times p}$ is the so called *seed matrix* and p is the number of derivatives computed. Hence, $p = n$ computes the full Jacobian with a single evaluation of f, $\left[\dot{\mathbf{y}}_1 \cdots \dot{\mathbf{y}}_p\right] = J_f \left[\dot{\mathbf{x}}_1 \cdots \dot{\mathbf{x}}_p\right]$. Each derivative update

is a vector operation and, in C++ codes, it can be implemented as a loop over each element of $\dot{\mathbf{x}}$.

2.1.2 Reverse mode

The RM, on the other hand, uses the adjoint formulation [38]

$$\bar{\mathbf{x}} = \frac{df}{dx}^T \bar{\mathbf{y}} \,. \tag{2.11}$$

$\bar{\mathbf{y}} \in \mathbb{R}^m$ (also called weight vector) is the vector for the adjoint direction and $\bar{\mathbf{x}} \in \mathbb{R}^n$ is the result of the adjoint formulation.

The RM propagates the adjoints — derivatives of the final result w.r.t. intermediate variables — in reverse order through the program flow. Consequently, the program is divided into two phases

$$\begin{aligned} &\text{for } i = n+1, \ldots, n+p+m-1 : \\ &\quad v_i = \Phi_i(v_j)_{j \prec i} && \text{(2.12a)} \\ &\text{for } i = n+p, \ldots, 1 : \\ &\quad \bar{v}_j = \sum_{j:j \prec i} \frac{\partial \Phi_i}{\partial v_i} \bar{v}_i && \text{(2.12b)} \end{aligned}$$

with initial values $v_{1\ldots n} = x_{1\ldots n}$, and adjoint values $\bar{\mathbf{v}}$ associated with the primal values. The required intermediate values for the *reverse section* (2.12b) are recorded during the *forward section* (2.12a) on a data structure called *tape*. Naively, without any data dependency analysis, all independent and dependent values as well as all intermediate variables have to be stored, as they can be arguments of the reverse section computation.

Seeding Seeding, in contrast to the FM, is done on $\bar{\mathbf{y}}$. The Jacobian is computed row-wise by setting $\bar{\mathbf{y}}$ to each of the Cartesian basis vectors $\mathbf{e}_i \in \mathbb{R}^m$. To compute the full Jacobian of f, RM has, thus, a time complexity of $\mathcal{O}(m) \cdot t_f$. Unlike the FM, gradients of f with $m = 1, m \ll n$, see (2.2), are computed efficiently with RM, at a constant cost factor independent of n. It can, however, be that the RM is computationally faster than the FM for $n \approx m$ [61; 90]. The RM traces a function f once, and the tape is then reused for the m adjoint computations (assuming the full Jacobian is computed), hence, avoiding repeated costly evaluations of f.

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_k} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial f_j}{\partial x_1} & \cdots & \frac{\partial f_j}{\partial x_k} & \cdots & \frac{\partial f_j}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_k} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

After j tape evaluations with the RM

The RM computes (2.8) and its Jacobian (2.9) in a single sweep. The SAC form of the forward and backward section results in

$$\text{Forward Section} \left\{ \begin{array}{llll} \bar{v}_{1\ldots2} & = 0 & & \\ v_{1\ldots2} & = x_{1\ldots2} & & \\ v_3 & = \Phi_3 & = v_1 v_2 & \\ v_4 & = \Phi_4 & = \sin(v_1) & \\ v_5 & = \Phi_5 & = v_3 v_4 & \\ y & = v_5 & & \end{array} \right. \quad \left. \begin{array}{rl} \bar{y} & = 1 \\ \bar{v}_5 & = \bar{y} \\ \bar{v}_4 & = v_3 \bar{v}_5 \\ \bar{v}_3 & = v_4 \bar{v}_5 \\ \bar{v}_1 & = \cos(v_1)\bar{v}_4 \\ \bar{v}_2 & = v_1 \bar{v}_3 \\ \bar{v}_1 & \overset{+}{=} v_2 \bar{v}_3 \\ \bar{x}_{1\ldots2} & = \bar{v}_{1\ldots2} \end{array} \right\} \text{Backward Section} \quad . \tag{2.13}$$

In the forward section, on the left, the intermediate values are computed as normal. The backward section, on the right, the function is evaluated in reverse order to compute the adjoints, see (2.12b). Note: $\bar{v}_1$ is updated twice during the backward section, as the adjoint value $\bar{v}_1 = \bar{x}_1$ is influenced by the computation of the sin function (v_4) and the previous multiplication (v_3). After the evaluation of the backward section, $\bar{x}_{1,2}$ contains the final derivatives, i.e., the Jacobian entries, $J_f = [\bar{x}_1, \bar{x}_2]$.

Unlike the FM, the memory requirement for the RM is harder to quantify. A trace of the computation of f is required for the reversal of the computational path during the backward section. The memory complexity is proportional to the amount of elemental functions, as, in the worst case, all values are put on the tape since they can be overwritten in the forward section. Conditional branch execution, loops or recursion in codes influence the memory size of the tape, causing the memory requirements to be dependent on $\mathcal{O}(n + p + m)$. Since a priori the memory requirements are hard to quantify, and it is easy to run out of memory for seemingly simple codes, techniques exist to reduce the overhead, see Section 2.3.

Vector mode extension The RM requires m evaluations of f to compute the full Jacobian J_f, see (2.1.2). Similar to the FM, the overhead can be reduced by using the vector mode. The RM vector mode, thus, uses a vector $\bar{\mathbf{y}}_i \in \mathbb{R}^m$ for each output of the function f. This results in the general product $J_f^T W$ where $W \in \mathbb{R}^{m \times q}$. W is the weight matrix, like S for the FM, and q is the number of derivatives computed in a single sweep of f.

2.2 Implementation

Most AD implementations in the context of C++ are relying on overloading instead of source transformation. A brief introduction to source transformation is, nevertheless, given in Section 2.2.1 with a focus on the fundamental concerns thereof. Subsequently, the overloading technique is described in Section 2.2.2. The focus is on the overloading concept, and the introduction uses example implementations for the FM. For brevity, implementation details for the RM are omitted, the presented concepts, however, still apply.

2.2.1 Source transformation

The complexity of object oriented code in combination with template metaprogramming is one of the reasons why no generally applicable AD source transformation tool for C++ exists. The existing ones mostly target the C language and can, thus, deal only with a subset of C++. However, a whole function or a whole program view is only possible with source transformation tools, which may facilitate more compiler optimizations.

Source transformation AD tools act as a preprocessor to the target code. It is parsed, typically, to an Abstract Syntax Tree (AST) representation, and then several analysis passes are run before the final AD augmented code is produced [49]. The analysis pertains to data, alias and control flow analysis which help identify active variables (w.r.t. AD): (1) For the FM, not every statement may be required to be augmented with (2.7). (2) Likewise, for the RM, determining which variable is required in the backward section can reduce memory overhead. For instance, a variable may be overwritten in the forward section, hence, the original value must be regenerated in the backward section at a specific point.

However, with just a static view of the target code, inherently the alias analysis is rather limited for complex codes. This represents one main challenge for source transformation tools targeting C++, as identified by [50].

More recently, for C++, source transformation tools have been used in combination with overloading [80]. Source transformation is applied to a C-like subset of the program for efficiency and compiler optimizations and the other program parts use the overloading tool.

2.2.2 Operator overloading

C++ allows for the customization of the semantics of common (elemental) operators by introducing user-defined types, called operator overloading.[1] Using AD overloading, the AD type $\widetilde{T}$ replaces the floating-point type T of the target code. It overloads all relevant operations and mathematical functions. For each invoked overloaded operation, a new (temporary) object of type $\widetilde{T}$ with updated derivative values based on the executed operator is returned,

$$\widetilde{T} \circ \widetilde{T} \mapsto \widetilde{T} \,. \tag{2.14}$$

The operator $\circ$ represents the operation that is applied to combine the two values of type $\widetilde{T}$, e.g., a multiplication. This is similar to the SAC, see (2.5b). For the FM, this is done with, e.g., an additional derivative value that is encapsulated in $\widetilde{T}$ along the primal value of type T. The approach is sometimes referred to as *full encapsulation* [32]. The RM type, on the other hand, typically holds a reference to the *tape* and pushes the required gradient data onto the tape during the operator execution.

FM overloading exemplified The implementation of a (naive) overloading AD tool for the forward mode is shown in Listing 2.1.

The derivative computation is done by changing the type of the target code and executing it with the new type with a prior seeding, see Listing 2.2.

Expression templates – Efficient overloading Modern AD overloading tools use expression templates [134] for a more efficient approach to overloading on a statement level. With expression templates, each overloaded operator does not return a temporary AD type $\widetilde{T}$ object but rather a generalized expression object `Expr`,

$$\mathrm{Expr}_A \circ \mathrm{Expr}_B \mapsto \mathrm{Expr}_{A \circ B} \,. \tag{2.15}$$

[1] https://en.cppreference.com/w/cpp/language/operators

```
1  class adouble  {
2   double p; // primal value
3   double d; // derivative value
4  public:
5   adouble(double primal, double deriv=0.0) : p(primal), d(deriv) { }
6   void seed(double s_v) { d = s_v; }
7   double derivative() const { return d; }
8   double primal() const { return p; }
9  // multiplication operator overload:
10  adouble operator*(const adouble& other) {
11    return adouble(/*primal =*/ p*other.p,
12                   /*deriv. =*/ p*other.d + d*other.p);
13  }
14 };
```

Listing 2.1: A minimal FM overloading implementation called `adouble`. The multiplication is overloaded, it returns a new object with updated derivatives. Other operators are implemented equivalently and yield no further insight.

```
1  double foo(double x) {
2   return x*x*x;
3  }
4  void bar() {
5   double x = 3.0;
6   double y = foo(x);
7   std::cout << y
8             << std::endl;
9  }
10
11
```

```
adouble foo(adouble x) {
 return x*x*x;
}
void bar() {
 adouble x = 3.0;
 x.seed(1.0);
 adouble y = foo(x);
 std::cout << y.primal()
           << y.derivative()
           << std::endl;
}
```

Listing 2.2: On the left, the original target code is shown. On the right, the `double` type is replaced by the `adouble` type, and before executing `foo` seeding is done.

The introduced expression type `Expr` stores information about the applied operation. The combining of sub-expressions is done on a statement level, hence, long statements lead to deeply nested compound expression tree objects. The advantage to the simple overloading approach is that an optimizing compiler can evaluate these expressions on a statement level, avoiding the creation of temporaries (on the tape). The expression trees are eventually resolved when they are assigned to a value of type $\widetilde{T}$ on the left-hand side. The gained computational efficiency, and the memory savings by avoiding temporaries on

the tape can be significant, as shown in [56].

In Listing 2.3 the expression template concept is applied to the example `adouble` FM class of Listing 2.1. Note that the presented implementation is not optimal and only for illustration, see [7; 111] for an extended discussion. The general concepts, however, apply. It makes use of template metaprogramming concepts, e.g., static polymorphism using the *Curiously Recurring Template Pattern*, see [4]. For brevity, only the main concepts of the AD hierarchy are explained:

Expression The class `Expr` is introduced and represents a proxy object and static interface for the expressions in a statement (line 1–5). It encapsulates the abstract type `T` which can be another compound expression or the scalar `adouble` class.

MulExpr The multiplication is not done directly on the `adouble` object (cf. Listing 2.1), but is defined as a proxy class that inherits from the expression class (line 7–21). The object is constructed with the abstract expression objects for the left and right side operands of the multiplication (line 12–13). To calculate either the primal or the derivative value, the respective functions need to be explicitly invoked (line 14–20), hence, a delayed evaluation is now possible.

The multiplication of two expression objects is realized through a standalone function (line 23–26). It simply returns the proxy object used for the multiplication of two expression objects, i.e., $\text{Exp}_{A \cdot B}$, see (2.15).

adouble The `adouble` class, also, inherits from the expression class (line 28–41). It represents the leaf nodes of the expression tree objects. Multiplying two `adouble`, the compiler invokes the standalone function (line 23–26), which instantiates a `MulExpr` object. The API is almost identical to the naive overloading approach, except for the conversion constructor that accepts a generic expression object and triggers the computation of the primal and derivative value (line 37–40). Whenever an assignment happens, the compiler invokes the conversion constructor and the calculation is only then done.

```
1  template <typename T>
2  class Expr {
3  public:
4   const T& cast() const { return static_cast<const T&>(*this); }
5  };
6
7  template <typename L, typename R>
8  class MulExpr : public Expr<MulExpr<L,R>> {
9   const L& l;
10  const R& r;
11 public:
12  explicit MulExpr(const Expr<L>& a, const Expr<R>& b)
13   : l(a.cast()), r(b.cast()) { }
14  auto derivative() const {
15   return l.primal() * r.derivative()
16        + l.derivative() * r.primal();
17  }
18  auto primal() const {
19   return l.primal() * r.primal();
20  }
21 };
22
23 template <typename L, typename R>
24 auto operator*(const Expr<L>& a, const Expr<R>& b){
25  return MulExpr<L, R>(a, b);
26 }
27
28 class adouble : public Expr<adouble> {
29  double p; // primal value
30  double d; // derivative value
31 public:
32  adouble(double primal, double deriv=0.0) : p(primal), d(deriv) { }
33  void seed(double s_v) { d = s_v; }
34  double derivative() const { return d; }
35  double primal() const { return p; }
36
37  template<typename E> adouble(const Expr<E>& other){
38   p = other.cast().primal();
39   d = other.cast().derivative();
40  }
41 };
```

Listing 2.3: Expression templates applied to Listing 2.1. It implements the concept of a delayed evaluation of expression trees by creating proxy objects for the multiplication. Only during the final assignment in a statement it is evaluated.

The application of the extended `adouble` class to a multiplication expression, thus, generates an expression tree object, see (2.16). For each operation in a statement, a deeper nesting of the `MulExpr` is statically instantiated until the assignment operator is invoked, thus, terminating the recursion. An optimizing compiler can then more easily optimize the statements during compile time, eliminating, e.g., temporary objects that would have been generated otherwise.

$$\underbrace{\texttt{adouble y} = \underbrace{\underbrace{\texttt{x} * \texttt{x}}_{\texttt{MulExpr<adouble,adouble>}} * \texttt{x;}}_{\texttt{MulExpr<MulExpr<adouble,adouble>,adouble>}}}_{\text{Assignment: Calls } \texttt{derivative()} \text{ and } \texttt{primal()} \text{ on MulExpr}} \tag{2.16}$$

Expression templates can be used to optimize the derivative evaluation on a statement level. In particular, some expression template-based AD overloading tools implement the forward mode using the so called statement level reverse mode, e.g., see the AD tool ELRFad of the Sacado package [110; 112]. A statement in C++ can be interpreted as a function $f : \mathbb{R}^{\hat{n}} \mapsto \mathbb{R}$, i.e., on the left-hand side a scalar gets assigned the result of an expression (i.e., f) of "many" ($\hat{n}$) inputs. Thus, on a global program scale the FM with its particular memory and computational complexity is applied, but each local gradient of a statement is computed with the RM. This technique is also employed by source transformation tools, e.g., ADIFOR [15], which transforms the target code globally with FM but on a statement level, the tool can use RM transformations. This concept is known as *statement-level reverse*, see [45], or *assignment-level preaccumulation*.

Additionally, the same concept can be used to evaluate the static expression tree to reduce the amount of adjoints that are stored on the tape structure. Instead of putting temporary adjoint values on the tape, (1) the whole expression is evaluated and the local adjoint values of the whole statement are computed, recursively, through the expression tree, and, then, (2) the adjoints of only the input and output of the expression are put on the tape [56; 118].

While resulting in more efficient AD code, the above described expression template technique typically comes at the price of higher compile times and, also, larger binary sizes of the compiled target application [130].

2.2.3 Seed-Compute-Extract paradigm

The seed-compute-extract paradigm is a term describing the generalized, logical partition of AD enhanced codes into three different phases [108]. As shown in Listing 2.2, for the FM, (1) the seed phase pertains to setting the initial gradient values (seeding) of the AD variable, (2) the compute phase is the execution of the target function(s) with the AD type, and, finally, (3) the values and gradients are extracted (and further processed). In contrast, for the RM, the compute phase is replaced by using the tape that is generated by evaluating the target function once. A generalized form of this approach is shown with the FM and RM AD driver functions based on a mock AD overloading tool, see Listing 2.4.

The implementation of the seed-compute-extract concept in Listing 2.4 illustrates the minimum of additional code that is necessary for AD. The code needs to be generic and compatible with different type definitions for the computation, i.e., the built-in `double` data type is replaced by the AD overloading type. Typically, this is done by introducing an alias statement for the underlying floating-point type. The computationally relevant code sections are (made) generic to support the change of data types. On the full computational path, this alias is then used and facilitates the type switch for AD. This principle is not as straightforward, see Chapter 3, and frequently causes compiler errors, necessitating further changes to the target code, further discussed in Chapter 4.

2.3 Hierarchical view of algorithmic differentiation

A black-box application of FM or RM AD can lead to significantly slower code and, especially for the RM, excessive additional memory usage. For the most efficient use of AD, further optimizations have to be applied by the AD expert, going beyond a black-box view. Specifically for RM related optimizations, the reduction of memory consumption is often required for even small simulations to run within memory bounds of the target hardware environment. At the same time, runtime is significantly influenced by the tape size, on which store and retrieval operations are applied. Hence, strategies to reduce the overall tape size, like expression templates, reduce the runtime [34].

Reducing the complexity of the AD computation is gained through (1) an optimally implemented AD overloading tool, and (2) identifying abstraction hierarchies of the code and exploiting the associativity of the chain rule *in a suitable way*.

```
1  template <typename Function,
2            typename Vector,
3            typename Matrix>
4  void fm_J(const Function& f,
5            const Vector& x,
6            Matrix& J,
7            size_t n,size_t m) {
8    std::vector<adouble> g_x, g_f;
9    // ... init g_x = x, g_f = 0 ...
10   for (size_t i = 0; i < n; ++i) {
11     // 1. seed:
12     g_x[i].seed(1.0);
13     // 2. compute:
14     f(/*in=*/g_x, /*out=*/g_f);
15     // 3. extract:
16     for (size_t j = 0; j<m; ++j) {
17       J(j,i)=g_f[j].derivative();
18     }
19     g_x[i].seed(0.0);
20   }
21 }
22
23
24
```

```
template <typename Function,
          typename Vector,
          typename Matrix>
void rm_J(const Function& f,
          const Vector& x,
          Matrix& J,
          size_t n,size_t m) {
  std::vector<adouble> g_x, g_f;
  // ... init g_x = x, g_f = 0 ...
  tape.setActive();
  // ... register g_x on tape...
  f(/*in=*/g_x, /*out=*/g_f);
  tape.setPassive();
  // ... register g_f on tape ...
  for (size_t i = 0; i<m; ++i) {
    g_f[i].seed(1.0);
    tape.evaluate();
    for (size_t j = 0; j<n; ++j) {
      J(i,j)=g_x[j].derivative();
    }
    g_f[i].seed(0.0);
    tape.clearAdjoints();
  }
}
```

Listing 2.4: Abstract driver functions for the FM (left) and RM (right) using some mock AD tool `adouble`. *FM*: The function `f` is executed `n` times for the full Jacobian (line 10–21). After each invocation, a column of the Jacobian is extracted from the function result `g_f` and the seed value is reset subsequently (line 16–19). *RM*: A tape structure is used to trace the function `f` once. The input of `f` is registered on the tape prior to mark the start of the adjoints on the tape (not shown) (line 10–12). After the function evaluation, the tracing is stopped and the output is registered to mark the end on the tape (line 13–14). The tape is evaluated m times by seeding each output of `f` (line 16–17), the resulting derivatives are stored row-wise on the Jacobian (line 18–20), and, finally, the computed adjoints are cleared for the next iteration.

Modern AD tools make use of the aforementioned expression template technique to reduce temporaries and inline the overloaded operations. This optimization is given to the AD user “for free”, it requires no additional changes to the target code beyond what a standard

overloading tool requires.[2]

The exploitation of the chain rule associativity requires the AD expert to manually interact with the target code, and, consequently, requires detailed knowledge of it. The code is viewed beyond a chain of elementary operations, and this is known as the hierarchical approach to AD [14]. Code structures of interest start at the level of expressions and statements, which expression templates handles optimally as shown. Levels beyond are basic blocks (e.g., a sequence of chained statement without control flow), loops and entire (sub-) functions.

2.3.1 Dividing composite functions

Commonly, complex simulation codes make heavy use of external libraries for the simulation runs, which complicates AD-enhancing a target code as experience has shown [120]. Overloading AD tools allow for hybridization, i.e., the coupling of libraries for AD by manually providing the required library-specific derivatives, or code regions for which the derivatives can be derived symbolically. These manual derivatives have to be provided at appropriate points during the execution. In particular, as mentioned, linear system solvers can be treated as elemental operators in AD, as described in [14; 39]. Special considerations are needed for iterative linear system solvers w.r.t. convergence, see [2].

Here, the chain of calls of (2.3) can be viewed at different levels of granularity. Suppose in (2.3) a subset $f^{K-S} \circ \ldots \circ f^{K-T} \subseteq f$ are the elemental functions executed in a direct linear solver. In a hierarchical view, the functions are combined, $f_S = f^{K-S} \circ \ldots \circ f^{K-T}$. Suppose further, the differentiation of f_S is expensive to tape with the RM. Instead of taping the solver invocation, f_S can be interpreted as an elemental operator on a matrix and vector level w.r.t. AD with a known derivative J_S [105].

"Mind the gap" on the tape To that end, for the RM, f_S is a part of the code that is not taped in the forward section, which causes a gap in the tape at that specific point of the execution. The AD expert has to close this gap [105; 106] by manually providing the (hand-written) derivatives $\bar{f}_S$ for the reverse section. Most AD tools provide a so-called *external function interface*. It can be used to (1) push a reference to the implementation of $\bar{f}_S$ at the point of the invocation f_S including all required data. (2) Subsequently, only the primal implementation of f_S is called. (3) Finally, during the tape evaluation, the AD tool

[2] Exceptions may apply for some code constructs but are negligible overall, see Section A.1.1.

encounters the external function reference, causing the invocation of $\bar{f}_S$ with the provided data to close the gap on the tape.

2.3.2 Common techniques to exploit structure

All optimization techniques related to AD overloading require a manual interaction of the user with the overloading tool. They are applied to computationally expensive code regions of the target codes that have been previously identified. How to identify these *hot spots* is not always straightforward. Linear system solver (direct or iterative) in the code base are worthwhile candidates to treat specifically. Other candidates may not be as obvious, see Chapter 6 for an extended discussion of profiling for AD. Commonly applied techniques to exploit structure are:

Hybridization As described above, also see [90], the tape is directly accessed to manually provide known derivatives. This is done for performance reasons, or, if the source code is not accessible for AD overloading, see Section 2.3.1 for details. It can be seen as a generalization of the other described techniques, each of which require an AD expert to interact with the code region and AD tool API manually for optimizations.

Preaccumulation A particular flavor of hybridization, *preaccumulation*, as already described for expression templates, can also be applied at higher levels of the hierarchy. In the context of some function f^i for instance, one can mark the independents and dependents of f^i, pre-calculate and store only the Jacobian J_{f^i} on the tape during the forward section. This results in a reduced memory usage if the computational complexity of the function, and the intermediates that would be stored on the tape, is higher than precomputing and storing the Jacobian of f^i directly.

2.3.3 Managing the tape size

Managing the tape size for the RM is important. It can become a problem for large simulation runs to keep the tape in main memory.

Tape file IO Some AD tools [13; 137] can serialize the tape to disk during the forward evaluation. During the backward evaluation, the tape file is read in reverse order to the write operations to calculate the adjoints. While file IO should generally be avoided for efficiency, the access pattern of the RM adds another layer of complexity, as it goes contrary

to the typical read-ahead IO implementations for a forward direction. As a remedy, special AD-aware IO libraries can be used [138].

Checkpointing Before executing a known memory intensive regions, instead of tracing them, checkpoints are generated and the code is executed passively w.r.t. AD in the forward section, see [47; 51]. The missing adjoints of that region are computed *just in time*, using a previously generated checkpoint to start the tracing. This is, for instance, done for structures in the code iteratively computing a value relevant to AD, e.g., by checkpointing each iteration. It follows that there is a trade-off between the number of checkpoints and the computational overhead of the recomputation of that region for the adjoints in the backward section. Placing the checkpoints optimally is NP-complete [103], except for special cases, e.g., see [47]. Heuristics exist to improve upon finding optimality of the checkpointing scheme [91].

3 Case Studies: AD overloading tools in real world codes

The implicit promise of AD overloading is that overall a code base is structurally untouched after the AD-enhancement. However, the seed-compute-extract paradigm commands several changes to the target code, see Section 2.2.3. As it turns out, and is exemplified in this chapter, applying AD to non-trivial code bases often poses additional software engineering challenges beyond seeding and extracting the derivatives. This may impact the whole code base, forcing the AD expert to introduce many changes w.r.t. the newly introduced AD overloading type. These changes pertain to incompatibilities of the overloading AD type with common programming patterns, e.g., type conversions from the (previous) floating-point type to an integer type, say. Unsurprisingly, these complications can be found, to different degrees, in all the complex target codes. As such, they can be considered fundamental to the process of AD-enhancement.

This chapter is structured as follows. In Section 3.1, the target applications of the case studies [59; 61] are presented, and the main properties pertaining to AD overloading are highlighted. The common code modifications related to AD in target codes are highlighted in Section 3.1.3. In Section 3.2, these commonalities are juxtaposed with other HPC projects of similar scale. Finally, the chapter ends in Section 3.3 with a preliminary conclusion and motivation for the potential of tooling-based automation.

References In Section 3.1, AD related source changes of two case studies are discussed. The summary is partially based on the following publications.

- A. Hück, S. Kreutzer, D. Messig, A. Scholtissek, C. Bischof, C. Hasse. **"Application of Algorithmic Differentiation for Exact Jacobians to the Universal Laminar Flame Solver"**. In: *Computational Science - ICCS 2018*. Vol. 10862. Lecture Notes in Computer Science. Springer, 2018, pp. 480–486. DOI: 10.1007/978-3-319-93713-7_43.

- A. Hück, C. Bischof, M. Sagebaum, N. R. Gauger, B. Jurgelucks, E. Larour, G. Perez. **"A Usability Case Study of Algorithmic Differentiation Tools on the ISSM Ice Sheet Model"**. In: *Optim. Method. Softw.* 33.4–6 (2018), pp. 844–867. DOI: 10.1080/10556788.2017.1396602.

3.1 Case studies

The first study, see Section 3.1.1, focused on the initial type change and the required efforts to make the code ready for the user-defined AD type. The second code study presented in Section 3.1.2, on the other hand, has an already verified adjoint implementation. However, during a AD type exchange [59], (performance) bugs related to AD were revealed. In both cases, the AD overloading tool CoDiPack [117; 118] was chosen. It is developed for low overhead and general efficiency through modern template metaprogramming and other such optimizations.

3.1.1 Universal Laminar Flame Solver

The Universal Laminar Flame (ULF) solver [61; 142] is used for solving generic laminar flame configurations in the field of combustion engineering. The eventual goal for the AD type change in the ULF solver is to conduct parameter studies of the underlying physics. Initially, though, only the Jacobians required by the (external) solvers for the underlying Ordinary Differential Equation (ODE) are computed for verification with AD.

Code characteristics The relevant characteristics w.r.t. AD are (1) the C++ code base written in a mix of native C++ code and use of C library functions, (2) external library dependencies, e.g., solver libraries for the numerical integration which require the Jacobian of the state equation. Overall, the study can be viewed as an example of code modifications due to the introduction of user-defined types to an existing code base that is not optimally prepared for such changes.

Black box differentiation The AD type change was done in a black box approach [129], i.e., it does not discern between active and passive variables w.r.t. differentiation. For active variables derivatives need to be propagated, whereas it is not necessary for passive variables, whose derivatives always would be zero. However, this is currently not a concern

in ULF, as (1) the active region is confined to the state function, and (2) CoDiPack has mechanisms to reduce the tape size for the RM (minimizing the adjoints on the tape). Hence, a global type change was done by introducing an alias `ulfScalar`. Subsequently, all code sections were modified to make use of the new alias. In accordance to the software design in [108], a templatization of the core data structures was added. The data structures were instantiated using the alias.

Type-related problems Compilation errors with the new type occurred mostly at places where (1) variadic C library functions were used, (2) debugging macros (making use of these variadic functions), and, finally, (3) conversions from and to the external libraries which define their own internal data structures with the double type. Hence, further modernization efforts included type conversion and value extraction functions, which are specialized for the CoDiPack type. The variadic C library functions are wrapped using template parameter packs[1] in order to apply the aforementioned type conversion functionality. Likewise, the debugging statements needed some explicit treatment. For a more detailed discussion of the AD-enhancement process of ULF, see Section A.1.1.

3.1.2 Ice Sheet System Model

The second code, ISSM [83], in contrast, already has a validated AD implementation [84]. ISSM is a C++ ice flow modeling software developed by NASA/JPL. The code uses MPI for large scale distributed computations and employs external libraries for solving the resulting linear equation system of the ice model.

The goal for the adjoint version of the code is to run large scale parameter studies for verification and validation of the physical properties of the underlying model eventually. The existing implementation was based on the AD overloading tool ADOL-C. However, the tool introduces too much overhead for the aimed at scale of future (sensitivity) studies. From the perspective of the ISSM developers, the idea of introducing the more recent AD tool CoDiPack, which promises performance improvements, thus, seemed enticing.

Code characteristics Similar changes as with ULF are observed.

[1] https://en.cppreference.com/w/cpp/language/parameter_pack

Alias Definition. The separation of passive and active variables w.r.t. differentiation using two alias definition, `ISSMDouble` and `ISSMPDouble` respectively, is a distinctive feature for a code of this scale. The decision which code parts are passive w.r.t. differentiation can only be done by an expert. However, this approach can lead to less AD-related overhead overall.

Templatization. Templatization of data structures was done for similar reasons as described in the ULF case study. Changes include introduction of generic vector and matrix classes for dense or sparse data. Also, generic functions for value casting and (specialized) wrapper functions for heap memory allocations were added.

Linear System Solver. ISSM has bindings for two different solver libraries, which are integrated as external functions, see Section 2.3.1.

MPI Wrapper. ISSM wraps every required MPI routine in a centralized manner. Each MPI wrapper forwards the call to different routines depending on the compile-time configuration. In total, four different configurations of the wrapper can be enabled, (1) AD and MPI forwards the calls to the Adjoinable MPI library, (2) no AD and MPI forwards to plain MPI routines, (3) and (4) serial code (no MPI in all combinations) is handled by emulation of the MPI routines, i.e., local copy of the buffer.

Defects In the process of adding CoDiPack, and validating the modified ISSM code, two defects were identified. First, the original ADOL-C utilization was not optimal w.r.t. runtime. This became apparent at finer mesh resolution during testing. Second, the adjoint MPI library AMPI [121], used by CoDiPack, needed extension to fit the requirements of MPI communication in ISSM. For an extended discussion on these defects, see Section A.2.

3.1.3 Common code modifications in target codes

The two case studies have shown several common patterns of related code changes. Most of these changes do not fundamentally influence the existing core code structure but are more locally applied. Instead, they improve the code by introducing abstractions through templatization. This gives a higher degree of flexibility to chose appropriate underlying data types and structures. Main commonalities can be summarized as follows.

Type alias The alias definition for the floating-point types is the core part of these type changes. All AD related computations use this alias. ISSM, as

an exception, makes a distinction between passive and active type aliases.

Templatization Data structures are introduced or extended to be independent of the core data type used for the computations. Hence, the program is compatible with the built-in float type and also supports user-defined types.

Data conversion Data conversions, sometimes implicitly introduced by the compiler, are resolved by the AD developer with conversion function.

External libraries External libraries are handled individually. For the ULF solver, the external parts are not relevant to AD. Linear system solvers in ISSM, on the other hand, are integrated as external functions with a manual symbolic matrix level derivative formulation for the best efficiency. In both cases though, the parts interfacing with the external library need to have data conversion points, where the primal value is extracted and passed to the library.

3.2 Juxtaposition of codes: Challenges of AD augmentation

In this chapter, so far, two case studies were presented to highlight the necessary efforts when applying AD overloading to complex real world codes. Although the efforts w.r.t. AD appear high, (1) for ULF, the gained accuracy of the derivatives and, also, the performance benefits compared to the FD methods were shown in [61]. (2) ISSM, on the other hand, already had a working AD implementation. Nevertheless, AD was motivated as the only feasible technique to produce the derivatives necessary for ice sheet model validation [84].

Analysing other AD overloading augmented software packages of comparable code complexity, similar AD-related code patterns, see Section 3.1.3, can be identified:

SU2 CFD Solver. A Computational Fluid Dynamics (CFD) solver package [30] developed with modern C++ and a size of approximately 160k Lines of Code (LOC). It uses CoDiPack through a centrally defined type alias definition set at compile time to (1) a FM, (2) RM, or (3) the double type. The SU2 code is structurally compartmentalized. Access to CoDiPacks functionality is done through free functions defined in

a single Translation Unit (TU)[2]. The functions handle the specifics of the tool, hiding implementation details. This includes starting the trace and other more advanced AD functionality like preaccumulation of code sections. Similarly, linear system solvers are invoked through a central TU.

SU2 is of comparable complexity to ISSM from an AD-related requirement viewpoint, i.e., distributed computations with MPI and efficient linear system solver use. Compared to both case studies of this chapter, for instance, it (1) provides overloaded functions to access the primal value for use in the context of, e.g., type casts. Though, there is no standalone cast function, and some casts appear to be a manual combination of a C++ cast with primal value extraction beforehand. (2) Overloads for the C function `sprintf` for IO operations exist. However, unlike with ULF, template parameter packs are not used. Instead, a fixed amount of manual overloads are provided, i.e., up to 18 arguments for the variadic function are supported. (3) MPI communication is handled with a centralized wrapper, a similar approach to ISSM. (4) Also, linear system solvers, e.g., a conjugate gradient solver, are integrated as external functions for efficiency. In addition, the developers make use of preaccumulation throughout the source code in order to reduce the memory overheads of the adjoint computation.

OpenFOAM. A large scale CFD solver [72] with around 850k LOC. An AD-enhanced version of one of the OpenFOAM solvers exists which uses the AD overloading tool dco. The process to integrate dco took several years [116; 130]. The overall software design goal of OpenFOAM is to be highly abstract and object oriented. This is achieved by making heavy use of templatization and overloading throughout the data structure hierarchies.

Due to the source codes overall complexity several problematic code constructs w.r.t. overloading tools are present. These include handling of explicit type casts, some implicit conversions and also unions which are not compatible with the AD type without further changes. On the other hand, only few uses of variadic C functions exist, e.g., for printing to console, and none in the context of the active type.

In addition, OpenFOAM uses MPI for distributed computing. In particular, it uses a wrapper and only exchanges data with the MPI datatype for bytes. The communication top layer, using the wrapper, is responsible for the data conversions. For AD, a distinction was introduced, i.e., if an active type is used, the AMPI overload is

[2] TU is the combined input from which an object file is generated (source file, included headers etc.)

invoked, otherwise the standard MPI routine is used. The check is done by adding type information and checking for active types dynamically.

Like SU2, the different first party iterative linear system solvers are handled by the aforementioned symbolic derivative formulation through an external function interface for efficiency.

Trace CFD suite. A CFD simulator developed by the DLR Cologne [36], written in C. Based on [117][3], the AD augmented Trace code supports two different AD overloading tools, one of them is ADOL-C. Trace being a C code, similar problems compared to ULF are described, e.g., C library function usage for heap memory management and IO operations need to be treated for AD. Memory management is handled by a wrapper, which instead uses the C++ new and delete functions, due to ADOL-C AD types requiring constructor calls during allocation. IO operations, e.g., printf, are handled comparable to the treatment of ULF using template parameter packs. External libraries were encapsulated to a single TU to avoid related type conversions scattered throughout the code base.

3.3 Summary and outlook

So far, the discussion focused on the process of augmenting codes with AD and related problems w.r.t. (1) localized problematic code constructs, (2) handling of external dependencies, (3) MPI, and (4) efficiency, e.g., special treatment of linear system solvers or, employed in SU2, preaccumulation for reduced memory overheads. The two presented case studies, but also the work of others, show many of the same problems as well as solutions. The complexity and multi-faceted workload of augmenting codes with AD and the source level commonalities, thus, motivate the idea of automation through a compiler-backed tooling approach.

The problematic code constructs arise from the different treatment of the built-in floating-point type compared to the user-defined AD types. Template metaprogramming, facilitated by modern C++, can be seen in many of these codes. It helps the developer to deal with these constructs in a type-independent manner. The metaprogramming helps with type casts, primal value extraction, variadic functions and data structures for vector or matrix representation. This, in turn, also facilitates AD overloading type exchanges. The choice of the AD tool is rather significant as the performance gains with CoDiPack showed [59].

[3] The source code was not available.

Likewise, in the future, other improved AD tools may be similarly integrated into ISSM for a performance maintenance enabling higher simulation fidelities. A detailed discussion of these overloading related problematic code constructs, their root cause and solution with tooling can be found in Chapter 4.

MPI for the RM is handled with one of the existing adjoint MPI libraries [119; 121; 131], depending on which bindings the AD overloading tool supports. However, as encountered with ISSM, (subtle) bugs may be present in these libraries. A discussion on MPI correctness facilitated by tooling is, therefore, done in Chapter 5.

The differentiation of external libraries is avoided if possible, due to the additional complexity w.r.t. development effort and induced computational overheads. In some cases, see ULF, data conversions are sufficient to reintegrate the external library into the AD augmented code. Other times, symbolic hybrid differentiation is done to integrate, see ISSM, SU2 and OpenFOAM. For SU2 especially, preaccumulation is used for suitable code locations to reduce the overall size of the tape. In Chapter 6, a framework for AD domain-specific profiling of applications to detect optimization opportunities is discussed.

4 OO-Lint: Enabling operator overloading in codes

In the previous chapter, aspects of the augmentation of complex target codes with an AD overloading tool have been described. The AD tools fully encapsulate the primal and derivative value (or a tape reference for the RM) [32]. To that end, all required operators are overloaded, thus defining the principle interaction of the target code with the AD tool. As a result, the code stays unchanged except for the seeding and extraction of the derivatives. While the semantic augmentation of codes with AD overloading is conceptually straightforward, the underlying type change leads to several complications which can cause compilation errors. This is due to the C++ language standard treating built-in floating-point types differently from user-defined types, i.e., the AD type. Well-formed codes in the context of the built-in can be in violation of the language standard after the type change. The amount of compiler error messages can be overwhelming, resulting in a tedious workflow of (1) identifying errors in the compiler output, (2) analysing the root cause and (3) subsequently fixing them.

This chapter identifies the root causes of errors and classifies them. This helps developers to (1) write new codes, using this analysis as a blueprint, for generic type compatibility, and (2) helps reason about problematic code constructs in legacy codes. The mechanical approach of resolving these issues motivated the implementation of a static code analyser OO-Lint based on the Clang compiler framework. It searches for patterns in the AST of a target code to automatically detect these issues. For each match, a diagnostic message is generated and, where possible, a fix via source transformation is applied.

The rest of this chapter is structured as follows: Section 4.1 presents an example of a problematic code construct extracted from OpenFOAM that causes a compiler error after the type change. In Section 4.2, additional problematic code constructs are presented, their root cause is analysed and localized code changes to fix these issues are discussed. A static C++ code analyser called OO-Lint based on the Clang compiler framework is

presented in Section 4.3. In particular, it aims to find these code locations, issue diagnostic messages and fix them automatically. The tool is applied to several scientific codes in Section 4.4 and the results are discussed. Finally, Section 4.5 concludes this chapter with a discussion, current limitations and related work.

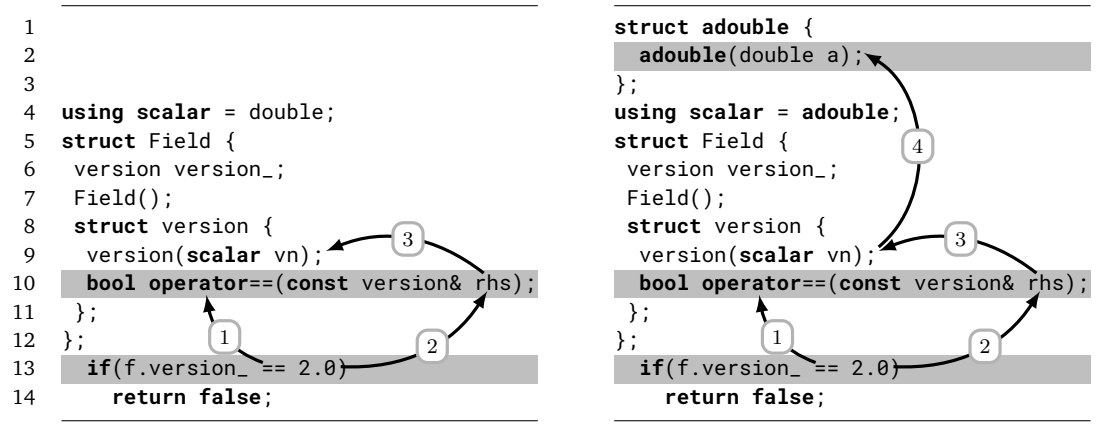
References The contents of this chapter are based on the below listed works. Additional aspects, not previously published, are the discussion of variadic functions in the context of overloading tools.

- A. Hück, C. Bischof, J. Utke. **"Checking C++ Codes for Compatibility with Operator Overloading"**. In: *15th IEEE International Working Conference on Source Code Analysis and Manipulation*. Vol. 15. IEEE, 2015, pp. 91–100. DOI: 10.1109/SCAM.2015.7335405.
- A. Hück, J. Utke, C. Bischof. **"Source Transformation of C++ Codes for Compatibility with Operator Overloading"**. In: *Procedia Comput. Sci.* 80 (2016), pp. 1485–1496. DOI: 10.1016/j.procs.2016.05.470.
- A. Hück, C. Bischof. **"Operator Overloading Compatibility for AD - A Case Study of Scientific C++ Codes"**. In: *AD 2016. The 7th International Conference on Algorithmic Differentiation - Programme and Abstracts* (2016), pp. 87–90. URL: www.autodiff.org/ad16/AD2016ProgrammeAndAbstracts.pdf.

4.1 Motivating example: OpenFOAM

In Figure 4.1 a simplified code excerpt from the OpenFOAM code base is shown. The code excerpt is an example for an implicit conversion that is legal code when the global type alias is set to the built-in type `double` but becomes ill-formed after the type change. In general, an implicit conversion is introduced by the compiler whenever a given type is different to the destination type in some particular context, but there is a conversion to the destination type that the compiler can apply to make the code well-formed again. However, in this particular example, after the type change, the compiler is required to apply two such implicit conversions on the literal value. The standard states in [67], §12.3-4:

> "At most one user-defined conversion [...] is implicitly applied to a single value."



(a) Implicit conversion.

(b) Ill-formed two-step implicit conversion.

Figure 4.1: Erroneous user-defined implicit conversion with a version check applied with a `Field` object (line 13). A compiler perspective: (a) The overloaded comparison operator of the `version` object is called (1). For the right-hand side, the literal operand needs to be converted (2). The conversion implicitly invokes the version struct constructor (3), giving a single step implicit conversion chain: `2.0` $\mapsto$ `version`. (b) Before constructing the version object, the literal must be converted to an `adouble` (4). There are now two user-defined conversions applied to the same value: `2.0` $\mapsto$ `adouble` $\mapsto$ `version`.

Hence, without any further code changes except for the redefinition of the global alias, the type change caused the code to be ill-formed. As a result of the type change, the developer has to deal with thousands of lines of error output related to this and related problems, see Figure 4.2.

A normal compilation produces approximately $13{,}000$ lines of compiler output for OpenFOAM (version $1.6.0$-ext). After a type change to an AD type, close to $910{,}000$ lines were generated, mostly consisting of errors related to the type change. Searching this output quantity for the error locations, fixing one, recompiling to find the next cause is tedious. In addition, many of the error messages can be traced to a few root causes in the code, which are propagated through the inclusion of headers. Automatically finding and fixing these issues frees development time of this rather mechanical maintenance work.

```
In file included from /home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/Field.H:367:0,
  from /home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/labelField.H:40,
  from /home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/primitiveFields.H:38,
  from /home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/pointField.H:37,
  from /home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/edge.H:41,
  from /home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/edgeList.H:33,
  from /home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/primitiveMesh.H:58,
  from /home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/polyMesh.H:45,
  from ../decompositionMethods/lnInclude/decompositionMethod.H:39,
  from lnInclude/scotchDecomp.H:39,
  from engineScotchDecomp/engineScotchDecomp.H:50,
  from engineScotchDecomp/engineScotchDecomp.C:27:
/home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/Field.C: In constructor‘
Foam::Field<Type>::Field(const Foam::word&, const Foam::dictionary&, Foam::label)’:
/home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/Field.C:256:33: error: no match for‘
operator’== in ‘is.Foam::IOstream::version() == 2.0e’+0
/home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/Field.C:256:33: note: candidates are:
/home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/IOstream.H:157:22: note: bool
Foam::IOstream::versionNumber::operator==(const Foam::IOstream::versionNumber&)
/home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/IOstream.H:157:22: note: no known
conversion for argument 1 from ‘’double to ‘const Foam::IOstream::versionNumber’&
/home/ahueck/OpenFOAM/OpenFOAM/src/OpenFOAM/lnInclude/VectorSpaceI.H:646:13: note: template<class
Form, class Cmpt, int nCmpt> bool Foam::operator==(const Foam::VectorSpace<Form, Cmpt, nCmpt>&, const
Foam::VectorSpace<Form, Cmpt, nCmpt>&)}
```

Figure 4.2: Compiler generated error message due to two implicit conversions applied to the same value illustrated in Figure 4.1 (b).

4.2 Semantic augmentation with overloading

In this section, the caveats of the AD overloading approach are first discussed. A motivating, first example was already given in the previous Section 4.1. For each such problem, a separate discussion using abstract code examples is provided in Section 4.2.1, before solutions for these problems are presented in Section 4.2.2. The proposed code fixes include distinctions between legacy and more modern C++ language standards.

Notation The notation in this section is based on the one used in Chapter 2: `scalar` is an alias for a built-in type T, or a user-defined $\widetilde{T}$ after the type change. U is some other type. For the code examples $T \equiv$ `double` or $\widetilde{T} \equiv$ `adouble` is used interchangeably. `adouble` represents some user-defined type, not limited to an AD overloading type. The discussion of this chapter is valid for all kinds of user-defined types.

4.2.1 Problematic code constructs

Typically, with the augmentation of codes using overloading, as shown in the context of AD in Chapter 3, the built-in type T is replaced by the user-defined type $\widetilde{T}$. The introduction of $\widetilde{T}$, however, is causing compile time errors with certain code constructs. These *problematic* code constructs are discussed in this section. Necessary source code changes to remedy these defects in the context of $\widetilde{T}$ are subsequently discussed in Section 4.2.2.

In particular, the following sources of errors are analysed:

Implicit conversion is a conversion from one type T to another type U *without* any explicit type conversion statement in the code, see Section 4.1. In the C++ standard, the implicit conversion of user-defined types must be performed in a single step.

Implicit bool conversion is a subset of implicit conversions, i.e., a conversion from T to a bool type. The compiler introduces these for arithmetic types in a conditional statement, evaluating to `false` if the value is 0 and otherwise to `true`.

Explicit conversion is a conversion from one type T to another type U *with* an explicit conversion statement, or an explicit construction using the copy constructor.

Unions are a type of class where member objects share the same memory region. The C++ language standard evolved over time w.r.t. allowing complex user-defined type members. Whereas they are basically disallowed in the C++03 standard ([66], §9.5-1), these restrictions have been relaxed in C++11 ([67], §9.5-2), allowing them to be members.

Name lookup ([67], §3.4) is the process of finding the corresponding declaration of a name encountered in a program, e.g., a function call and its corresponding declaration need to be matched. The process of looking up an overloaded mathematical function by $\widetilde{T}$ may become ill-formed when the compiler can not disambiguate between the overload and an existing implementation in the code base.

Variadic arguments used to pass different numbers of arguments to a function. They are a recently added analysis target, as a consequence of problems identified with the ULF solver, see Section 3.1.1. Variadic arguments are defined in the C language[1] but are explicitly supported by C++ for backwards compatibility ([67], §18.10). In Section A.1.1, the C library function `printf` for console output was such an example. These common C functions are not compatible with $\widetilde{T}$.

[1] https://en.cppreference.com/w/c/language/variadic

Implicit conversion In many instances, a C++ compiler automatically adds implicit type conversions based on contextual analysis. Especially arithmetic conversions between *fundamental types* ([67], §3.9.1) are common w.r.t. arithmetic expressions with different operand types. For instance, the so-called standard conversion between, e.g., *floating-point* and *integer types* are legal.

In the context of a user-defined data type $\widetilde{T}$, an implicit conversion on $\widetilde{T}$ is a *user-defined conversion* ([67], §12.3.1). The C++ standard restricts ([67], §12.3-4): "At most one user-defined conversion [...] is implicitly applied to a single value.". The code can become ill-formed if two conversions had been applied in sequence for type coercion to the original type which, after a type change, is now a user-defined type.

In Listing 4.1, the function `foo` accepts a reference to an object of class `X`. The argument of the call to `foo`, an integer literal, is converted to be of the user-defined type `X`. Although two conversions are applied on the literal value, only the last step counts as a user-defined conversion. Hence, the code is well-formed.

```
1  using scalar = double;
2  class X {
3  public:
4    X(scalar i);
5  };
6  void foo(const X& x);
7  void bar() {
8    foo(1); //OK: Conversion literal ↦ double ↦ X: foo(X(double(1)));
9  }
```

Listing 4.1: Implicit conversion with a fundamental type.

In Listing 4.2 the same code is shown after a type change to a user-defined type.

The class `adouble` adds new semantics to the previous program. It replaces the parameter of the constructor of class `X`. Now, the compiler generates an error for the call to the function `foo`. The value `1` has to be transformed to `adouble` before being passed to the constructor of `X`. This represents two *user-defined conversions* on the same value and the code is thus ill-formed.

Implicit bool conversion Implicit bool conversions ([67], §4.12) of, e.g., arithmetic types are applied in contexts where a bool value is expected. In Listing 4.3, the condition of the if statement is such a context. The value of the double is interpreted as a bool value,

```
1  class adouble {
2  public:
3    adouble(double i);
4  };
5  using scalar = adouble;
6  class X {
7  public:
8    X(scalar i);
9  };
10 void foo(const X& x);
11 void bar() {
12   foo(1); //Error: Two user-defined conversions literal ↦ adouble ↦ X
13 }
```

Listing 4.2: Ill-formed two step implicit user-defined conversion.

and the branch is executed for a non-zero value. A type change makes the code ill-formed as the compiler is not free to evaluate the new type as a bool.

```
1  using scalar = double;
2  void foo() {
3    scalar a = 1.0;
4    if(a) {}
5  }
```

Listing 4.3: A bool conversion inside the condition of an if statement.

Explicit conversion An explicit type conversion is commonly called *casting*. The developer explicitly states the intent to transform one data type into another with a cast statement. Several cast expressions are supported in C++. This includes the C-style cast expressions `(type) expression` or `type(expression)` ([67], §5.2.3 and §5.4, respectively). The latter is also called *functional-style cast*. C++ introduced four cast operators in addition (see [67], §5.2):

- `static_cast<Ty>` is a standard type conversions, comparable to C-style casting.
- `reinterpret_cast<Ty>` is an unsafe type conversion (e.g., pointer conversions).
- `const_cast<Ty>` removes const properties of a pointer or reference.

- `dynamic_cast<Ty>` is a run-time checked conversion. It converts a pointer from one class to another, along the inheritance hierarchy.

Explicit conversions are problematic when they are applied to the user-defined class. Semantically, the underlying value of type $\widetilde{T}$ needs to be exposed for the cast operation. In Listing 4.4 an explicit cast expression is shown. After the type change to an AD type, say, the primal value needs to be accessed for the cast to be semantically equivalent.

```
1  using scalar = double;
2  void foo() {
3    scalar a = 1.0;
4    int value = static_cast<int>(a);
5  }
```

Listing 4.4: Explicit C++ type cast of a scalar value to an int.

Unions A `union` is a class type where members share the same memory region, and its size is determined by the largest member. The member access is done referencing the union's name. Unions can also be anonymous, each member is considered to have been defined in the enclosing scope of the anonymous union. An example is shown in Listing 4.5, where an integer and a double type share the same memory region.

```
1  using scalar = double;
2  union X {
3    scalar a;      // 8 bytes
4    int b;         // 4 bytes
5  };  // Whole union: 8 bytes
```

```
using scalar = double;
union {
  scalar a;
  int b;
};
```

Listing 4.5: A named (left) and an anonymous union (right) with two members each.

A union member with a non-trivial (copy) constructor, destructor or copy assignment operator is prohibited in the C++03 and earlier language standard ([66], §9.5-1). As a consequence, it is impossible to use a `union` with a non-trivial overloading class $\widetilde{T}$. With the introduction of the C++11 standard, these restrictions are relaxed, sometimes called *unconstrained unions*. The developer, however, needs to define custom code extensions to the union for correct handling of the non-trivial type $\widetilde{T}$, see [67], §9.5-4:

> "Note: In general, one must use explicit destructor calls and placement new operators to change the active member of a union."

Name lookup The lookup of names of, e.g., function calls encountered by the compiler can be either unqualified or qualified using the scope resolution operator ([66], §3.4). Closely related to unqualified name lookups is the Argument-dependent Lookup (ADL) ([66] §3.4.2). ADL causes a lookup of a function call to consider its arguments for the search without requiring any namespace qualification ([123], §8.2.6). This is especially useful for overloaded operators. With ADL, when any overloaded operator is encountered by the compiler, and it is not found in the context of its use, the namespace of the operands are considered for the lookup [127].

Closely related to the concept of name lookup and ADL are the friend functions ([67], §11.3). They are used to accessing private members of a class, but they are themselves not members (not invoked on an object) and are not in the scope of the class. Hence, if a friend function is part of a class declaration body, but no outside declaration exists, only with ADL will the friend name be resolved.

Two problems regarding overloading are described, (1) the lookup of friend functions can fail in certain contexts, which pertain to the design of the AD overloading tool, and (2) lookups can become ambiguous when a user-defined type is introduced which defines its own set of overloaded functions.

In case (1), e.g., the tool ADOL-C provided a set of overloaded operators as friend functions without declaring them outside the class in the past. While they are resolved with ADL, qualified name lookups (e.g., `::sqrt(..)`) disable ADL which causes a failed name lookup. Hence, the program is ill-formed, see Listing 4.6.

```
1  class adouble {
2  public:
3    friend void bar(const adouble& x) {  }
4  };
5  namespace ns {
6    using scalar = adouble;
7    void foo() {
8      bar(scalar{});   // unqual. lookup, bar found with ADL
9      ::bar(scalar{}); // qual. lookup in global namespace fails, no ADL
10   }
11 }
```

Listing 4.6: A friend function inside the class definition. Qualified name lookup with the scope resolution operator disables ADL, the name is not found and the program is ill-formed.

In case (2), with name lookups in general, the overloading class can introduce ambiguities into the code. Typically, the user-defined type provides overloads of mathematical functions in addition to overloading all operators. To that end, the class and corresponding overloaded functions may reside in the global namespace. Thus, calls and operations w.r.t. $\widetilde{T}$ behave as if it were a plain floating-point type. However, as is the case with OpenFOAM, if a code defined its own implementations of, say, a mathematical function in its own namespace, a lookup might become ambiguous if $\widetilde{T}$ provides a corresponding overload. The compiler is unable to choose, causing the code to be ill-formed, see Listing 4.7.

```
1  adouble sqrt(const adouble& a);
2  namespace ns {
3    using scalar = adouble;
4    scalar sqrt(scalar v);
5    void foo() {
6      scalar a;
7      ::sqrt(a);   // OK: adouble sqrt(const adouble& a);
8      ns::sqrt(a); // OK: sqrt(scalar v), same as ::ns::sqrt(a)
9      sqrt(a);     // Error: Ambiguous call
10   }
11 }
```

Listing 4.7: The last call is ambiguous as the compiler finds two matching declarations.

Variadic arguments Variadic functions take a variable number of arguments, which, through a predefined set of macros, can be accessed ([67], §18.10). In Listing 4.8 two variadic functions are shown.

The left implementation is a function that computes the sum of an arbitrary number of values. It is fully compatible after the type change. On the other hand, commonly the C library functions for IO operations are used in codes, e.g., in the ULF solver for diagnostic output, see Section 3.1.1. These functions are black-box implementations and can only handle built-in types. For a user-defined type, the respective value that should be printed needs to be extracted explicitly.

4.2.2 Transforming problematic code constructs

In Section 4.2.1 several code constructs were introduced, which work with the built-in type but become ill-formed with the type change to a user-defined type. In this section,

```
#include <stdio.h>
#include <stdarg.h>
using scalar = double;
scalar sum(unsigned n, ...) {
  scalar sum = 0.0;
  va_list args;
  va_start(args, n);
  for (unsigned i = 0; i < n; ++i) {
    sum += va_arg(args, scalar);
  }
  va_end(args);
  return sum;
}
void foo() {
  scalar s = sum(2, 4.3, 3.4);
}
```

```
#include <stdio.h>
using scalar = double;
void foo() {
  scalar a;
  printf("%f\n", a);
}
```

Listing 4.8: *Left*: a sum function which uses variadic argument macros (`va_*`). *Right*: a C language `printf` function which is not compatible with user-defined types.

source transformations are presented to remedy this. The overall goal is to apply as little change as possible, hence the solutions focus on localized changes.

Implicit conversion Two possible approaches to transform the code to avoid the complications of multiple user-defined conversions on the same value are presented: (1) Completely disallowing implicit conversion, and (2) removing one step of the conversion chain.

User-defined conversions can be completely avoided by disallowing the compiler to introduce them in any context. This is achieved by using the C++ keyword `explicit` to each relevant constructor of user-defined types in the target code, available since the C++11 language standard. Implicit conversions are thus prohibited and will cause compile-time errors. They need to be explicitly applied by the developer.

Using `explicit` constructors will require extensive maintenance of the target code. Alternatively, introducing a functional cast to $\widetilde{T}$ at the code location of the implicit conversion also avoids the error. Unlike with the first approach, this is a localized change without any effect on other code regions, see Listing 4.9.

```
1   class adouble {
2   public:
3     explicit adouble(double i);
4   };
5   using scalar = adouble;
6   class X {
7   public:
8     X(scalar i);
9   };
10  void foo(const X& x);
11  void bar() {
12    foo(scalar(1));
13  }
```

Listing 4.9: The constructor is made `explicit` which forces the developer to manually convert to the `adouble` class (line 3). A functional cast removes one step of the implicit conversion chain, making it well-formed after a type change (line 12).

Implicit bool conversion Expression with bool conversions related to the type T will be ill-formed after the type change as there is no conversion rule for user-defined types in these contexts.

There exist so-called conversion functions which allow user-defined conversions to, e.g., bool values in the appropriate context. However, the usage of these are not advised [124]. The compiler is free to use them in unforeseen contexts such as any arithmetic expressions or comparison of unrelated objects in bool tests.

Due to these complications, making these implicit bool tests explicit is preferred, see Listing 4.10.

```
1   using scalar = adouble;
2   void foo() {
3     scalar a = 1.0;
4     if(a != 0.0) {}
5   }
```

Listing 4.10: The explicit bool comparison is semantically equivalent.

Explicit conversion Explicit cast operations are intended by the developer, and any change, therefore needs to retain its intended semantics. For an AD overloading type in particular, retaining the semantics means accessing and casting the primal value.

A template cast function like Listing A.3 can be used for these conversions. Alternatively, the C++11 standard allows user-defined types to have so called explicit user-defined conversion functions ([67], §12.3.2). They allow for the user-defined class type to be type cast to another, specified type. These operators allow the compiler to invoke them in the context of C++ cast operations for the type conversions. This is unlike the previously mentioned conversion function in the context of, say, if condition statements. The keyword `explicit` disallows the compiler to use them for implicit conversions, hence, it works only in contexts of explicit type conversion statements. However, certain user-defined conversion functions are applied by the compiler in specific contexts. For instance, the `if`-condition statement is such a context. Here, an explicit bool or generic template conversion function works. However, for older C++ standards, an implicit (bool) conversions would be required in these contexts without applying further code modifications, as described in the previous paragraph.

The overall merit of the template cast functions is the independence of the provision of these explicit conversion operators by the user-defined type. If the augmented target software, for instance, relies on the conversion function, switching to a different AD tool without such functionality may be hindered in the future. In contrast, the template conversion function can be easily extended to handle a different AD tool API. In Listing 4.11 both approaches are nevertheless illustrated.

```
1   class adouble {
2   public:
3     template<typename T>
4     explicit operator T() const;
5   };
6   using scalar = adouble;
7   void foo() {
8     scalar a = 1.0;
9     //calls "a.operator T", T ≡ int
10    int value = static_cast<int>(a);
11  }
```

```
#include "recast.h"

using scalar = adouble;
void foo() {
  scalar a = 1.0;
  int value = recast<int>(a);
}
```

Listing 4.11: Left: the user-defined type provides a (templatized) conversion function which leaves the C++ cast statements unchanged. Right: the recast function is part of the target project and is specialized for the user-defined type.

Unions Unfortunately, there is no generally valid transformation for unions to regain compatibility with user-defined types as the usage of unions can have multiple reasons. The question of why a union is used needs to be answered to formulate a transformation strategy that retains the semantics. If unions are solely used as a memory saving strategy, so called type safe unions introduced in a newer standard may be used instead. On the other hand, unions are often found in the context of *type punning*. It is the interpretation of the union's memory region with a different type. In numeric codes, type punning is used to access the elements of a vector class with either an index or a named variable, say, x, y, z. In [117], the author describes changes to unions for valid for C++11 and beyond to handle user-defined types in the context of type punning. However, while in the C language such usage is common, it is considered undefined behaviour in C++ [125].

Starting with the C++17 language standard, type safe unions called `variant` were introduced.[2] It has the same memory size semantics of an (unsafe) union, but offers additional safety w.r.t. the active member. There exist backward compatible non-standard implementations of `variant` and, as such, it is the proposed solution for the problematic named union, see Listing 4.12. Unfortunately, anonymous unions require a different approach as there are no equivalent anonymous variants.

```
1 #include <variant>
2 using scalar = adouble;
3 std::variant<scalar, int> X; // sizeof(X) equals sizeof(scalar)
```

Listing 4.12: A named union becomes a type safe variant.

Name lookup First, a solution for the problematic friend function usage, see Listing 4.6, is presented in Listing 4.13. A declaration of the friend is added outside the user-defined class. This allows the compiler to find the declaration without reliance on ADL.

The ambiguity of name lookups, see Listing 4.7, is generally not resolvable without knowledge about the underlying design principles of providing namespace qualifications. Nevertheless, two possibilities of resolving the ambiguity are presented. The notation for the discussion is as follows: Any function that is overloaded for the type $\widetilde{T}$ and present in the inner namespace `ns` is called φ. It takes as argument types either T or $\widetilde{T}$.

One solution is the removal of all definitions of φ in `ns::`. Consequently, the qualified (`ns::` or `::ns::`) function invocations have to be transformed to unqualified lookups.

[2] https://en.cppreference.com/w/cpp/utility/variant

```
1   class adouble {
2   public:
3     friend void bar(const adouble& x) {  }
4   };
5   void bar(const adouble& x);
6   namespace ns {
7     using scalar = adouble;
8     void foo() {
9       bar(scalar{});
10      ::bar(scalar{}); // qual. lookup in global namespace works
11    }
12  }
```

Listing 4.13: After adding a declaration of the friend outside the class body, lookup works and the program is well-formed.

This will cause the compiler to choose the overloaded function in the global namespace (with ADL). A caveat of this solution are calls to φ from within a method declaration which belongs to a class that also defines a method with the same name as φ, see Listing 4.14. Thus, for every instance of a call to φ inside a method of such a class, the call has to be changed to a qualified lookup in the global namespace, i.e., `::`φ.

```
1   namespace ns {
2     class X {
3       scalar φ(const X&);
4       void foo() {
5         scalar a;
6         φ(a); // Error: X::φ(const X&) is only candidate
7       }
8     };
9   }
```

Listing 4.14: Errorneous change: `ns::`φ is replaced by the unqualified lookup φ.

Another approach is the full qualification for each call to φ. It is preferable as it avoids the ambiguity described above. The full qualification requires precise knowledge about the location of the callee, i.e., in which namespace the name reside as ADL is disabled for these calls.

Variadic arguments Solutions for handling variadic arguments are shown in Figure 4.15.

While strictly not necessary, the sum function of Listing 4.8 is transformed to templatized code, removing the dependency on C language constructs and giving additional type safety. Two different implementations are given, one for C++17 and one for the earlier language standard C++11. The C library functions, on the other hand, have to be wrapped, see Listing A.4. The wrapper is implemented with template parameter packs to apply a function that extracts the underlying value of the user-defined type to be printed.

```
1   using scalar = adouble;
2   // C++17 fold expression:
3   template<typename... Args>
4   auto sum(const Args&... a) {
5       return (a + ...);
6   }
7   // C++11 parameter pack:
8   template<typename T>
9   auto sum_t(const T& t) {
10    return t;
11  }
12  template<typename T, typename... Args>
13  auto sum_t(const T& t,
14             const Args&... args) {
15    return t + sum_t(args...);
16  }
17  void foo() {
18    scalar s = sum(4.3, 3.4);
19  }
```

```
// for oo_printf(..):
#include "recast.h"
using scalar = adouble;
void foo() {
  scalar a;
  oo_printf("%f\n", a);
}
```

Listing 4.15: *Left*: The sum function is transformed to type-safe template code. *Right*: A wrapper function for `printf` is introduced which handles the extraction of the underlying value of $\widetilde{T}$, see Listing A.4.

4.3 OO-Lint static code analyser

The resolution of the problematic code constructs in Section 4.2.2 is rather mechanical, and at the same time the transformations are localized to the TU where the potential error occurs. This motivated the creation of a static analysis and transformation tool called OO-Lint based on the Clang compiler framework. In particular, with Clang, stand-alone tools can be developed that are applied to the AST of a target source file. The AST can be searched for patterns of the problematic source constructs and subsequently handled. The

tool searches for a certain type string in the code to identify relevant locations w.r.t. AD overloading. AST nodes which work on an alias like “scalar” or the built-in “double” are considered depending on the tool configuration.

4.3.1 The static analyser

Figure 4.3 shows the design of the tool. As input, the tool requires (1) the analysis target source file locations, (2) a compilation database [85], containing the exact arguments with which each source file is compiled, and (3) a configuration file for setting, e.g., the scalar type search string.

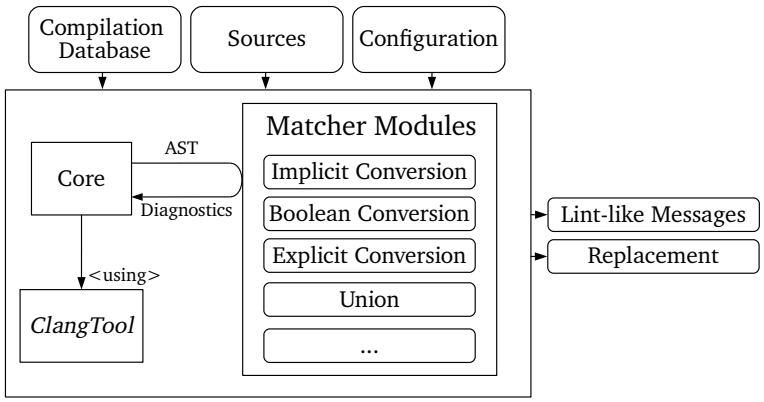


Figure 4.3: OO-Lint is based on the Clang tooling library, adapted from [60]. It defines matchers for each of the described problematic code constructs. A match generates a diagnostic message, and an optional source code replacement.

The compilation database is necessary for the creation of the AST. For instance, conditional include statements that are activated due to specific compilation flags may alter the structure of the code. The tool consists of different modules, each detecting a different problematic code pattern in the Clang AST. Each module produces Lint-like messages and optional source transformations (replacements) for the analysed file. They are collected and applied during post-processing.

4.3.2 Processing the Clang abstract syntax tree

The Clang AST is a high-level representation of the underlying C++ code. A node in the AST has a close relation with the textual representation in the source file. Additional nodes not in the source text are added by the semantic analysis phase of the Clang frontend, e.g., nodes for implicit cast expressions. To search the AST for a pattern, the so called Clang AST matcher interface can be used. A Clang matcher expression is assembled from a DSL-like matching language of Clang. The final expression is a pattern that is applied on the AST, and generates a callback for each match. The callback processes the match, extracting the source location to create the diagnostic message and relevant source replacements. The overall matcher workflow, which is implemented in OO-Lint, is shown in Figure 4.4.

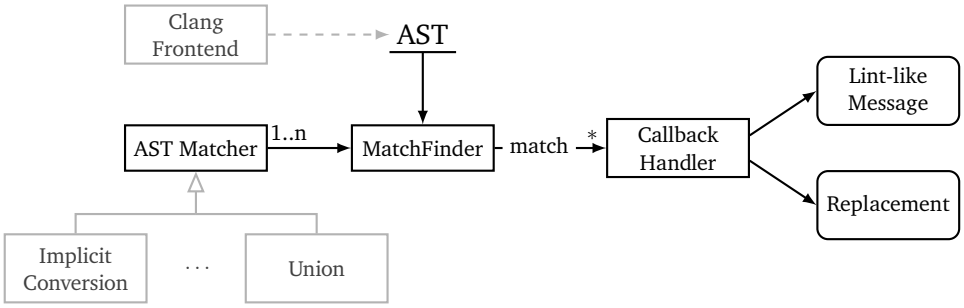
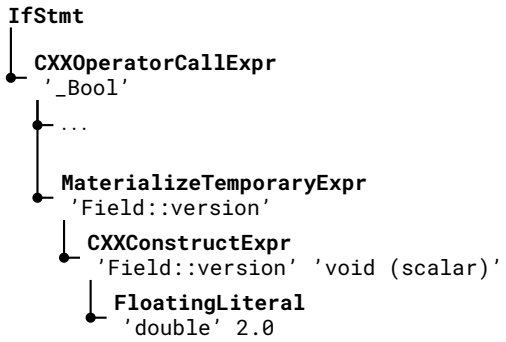


Figure 4.4: AST node matching. Clang generates the AST on which the AST matcher expressions are applied. For each generated match, a message and code replacements are generated.

In addition to the semantically annotated AST, the preprocessor phase of the AST creation process is used to collect location information of include directives. These are used to add required headers for, e.g., the `recast` function that replaces the C++ cast statements.

Matching Matching the AST with the Clang matcher expressions[3] is done by nesting different matchers to find nodes, traversals and attributes that represent the structure of the problematic code construct. In Figure 4.5, the simplified AST of the example given in Figure 4.1 is shown. The corresponding matcher expression finds these pattern, i.e., an instance of a problematic code construct.

[3] https://clang.llvm.org/docs/LibASTMatchersReference.html



(a) AST excerpt.

```
1 StatementMatcher conversion =
2  materializeTemporaryExpr(
3   hasTemporary(
4    ignoringImpCasts(
5     constructExpr(
6      hasImplicitConversion(type_s),
7      unless(temporaryObjectExpr())
8     ).bind("conversion")
9    )
10  )
11 );
```

(b) Matcher expression.

Figure 4.5: AST matching of implicit conversions, see Figure 4.1. OO-Lint searches for certain type strings `type_s`, e.g., “scalar”.

To that end, the matcher starts at the node `MaterializeTemporaryExpr` which is used in the context of implicit user-defined conversions. The traversal from the root node ignores other, unrelated implicit conversions casts and matches the child node `CXXConstructExpr`. This represents a call to a constructor — here, the `version` struct. Only objects that are created from a single explicit argument (not counting default arguments of a constructor [67], §8.3.6) with a mismatching argument and class constructor type are of interest. In the AST a temporary version object from a floating-point literal is created. This represents a mismatch of the expected type in the constructor with the actual argument (a literal) which causes a subsequent compilation error after a type change. A check for this is done with the matcher extension[4] `hasImplicitConversion`. We ignore nodes of the subtype `TemporaryObjectExpr` as they represent explicit functional casts instead of implicit constructor calls.

Source transformation A source transformation in Clang is based on string manipulations as the AST is inherently immutable. To that end, the tight coupling between AST nodes and the textual representation is used to specify text replacements. The location of implicit nodes points to the respective target expression, e.g., an implicit cast’s location points to the cast’s source expression. Source transformation operate with `Replacement` objects. They contain positional information and the respective replacement string. As such, the transformed source code file is unchanged except for the `Replacement` target

[4]Clang allows providing custom matcher extensions with its API.

strings. Macro code is not yet handled. Changing the underlying macro code can have unwanted effects in unrelated code sections. Likewise, inlining the code of the expanded macro at the problem location is not always desirable.

4.4 Evaluation

For the evaluation, the tool OO-Lint is applied to several scientific codes consisting of (1) OpenFOAM, (2) the ULF solver (prior to the AD augmentation), (3) SU2, (4) a special educational variant of SU2 called SU2_EDU, and, finally, (5) ISSM. For OpenFOAM and ISSM a separate, more thorough discussion is given at the end.

The scientific codes

Some analysed codes were discussed in detail in Chapter 3. A brief summary is given in the following.

OpenFOAM [72] is a C++ CFD solver package. It employs macros, templates and overloading throughout its code base of about $640{,}000$ LOC. The fundamental data type is defined using the alias `scalar`, which is set to the built-in `double` or `float` depending on compile time options. The report is based on version `3.0.x`.

ULF Solver [142] is a non-public solver for chemically reacting, laminar flows, written in C/C++. The analysed version predates the AD augmentation, hence, it makes no use of any special alias for the underlying floating-point type. The core ULF solver code base has about $60{,}000$ LOC.

SU2 is a CFD solver package written in C++ with about $80{,}000$ LOC. The code base uses the CoDiPack AD tool to compute derivatives [3; 141]. SU2 uses the `su2double` alias for the computations.

SU2_EDU is a trimmed down version of SU2 for educational purposes ($34{,}000$ LOC). It does not use the aforementioned alias and is not AD augmented.

ISSM [83] is a modeling software for ice flow developed by NASA/JPL. Notably, ISSM was made compatible with overloading AD with ADOL-C over a time frame of several years. The C++ code base consists of $80{,}000$ LOC. ISSM distinguishes between active and passive types w.r.t. AD using the aliases `IssmDouble` and `IssmPDouble`, respectively.

Analysis overview

Table 4.1 shows the analysis results of OO-Lint. For each code, the occurrences of problematic code constructs described in Section 4.2.1 are listed.

Most problematic code constructs are found in OpenFOAM. It is the only framework with the name lookup complications, see Section 4.2.1. The prototypical matcher for name lookup counts both name qualifications to the global namespace (e.g., `::sin`) and to the local namespace (e.g., `foam::sin`). The OpenFOAM namespace defines several transcendental functions. This may cause ambiguity with the AD tools overloads, discussed further below.

SU2 is inherently compatible with AD. The developers ensure compatibility for AD whenever relevant code parts are added. Unsurprisingly, no problematic code constructs are detected. In contrast, the EDU version uses plain double built-ins only as AD capabilities are omitted. Several problems are detected.

	OpenFOAM	ULF Solver	SU2	SU2_EDU
Translation Units	1992	125	109	25
Implicit Conversion	10	3	0	0
Implicit Bool Conversion	2	2	0	1
Explicit Conversion	35	25	0	5
Union	1	0	0	0
Name Lookup	307	0	0	0

Table 4.1: Result of the static analysis for OpenFOAM, ULF Solver, SU2 and SU2_EDU.

The analysis for variadic functions is a recent development in OO-Lint and foredates development of ULFs type change, and was, thus not applied to it. The other presented codes make little use of, e.g., `printf`. SU2 wrapped the usage thereof, SU2_EDU does not use them in a relevant context, same for OpenFOAM.

OpenFOAM OpenFoam is the only framework which has the name lookup complications described in Section 4.2.1. The framework uses the local namespace `Foam` and defines several transcendental functions (with macro expansion). These often simply forward the call to the global namespace, see Listing 4.16.

In addition, there is a mix of qualified and unqualified lookups, resulting in calls like `Foam::sin`, `::sin` or `sin` w.r.t. the AD type.

```
1  namespace Foam {
2    scalar sin(scalar x) { return ::sin(x); }
3    void somewhere() {
4      scalar a;
5      Foam::sin(a);
6    }
7  }
```

Listing 4.16: Transcendental function definition. These are generated through a macro expansion (not shown).

Reviewing the adjoint OpenFOAM 3.0 source code [130], the changes required mirror the proposal of Section 4.2.2. In the following some relevant changes are discussed.

- The transcendental functions and the name lookups were handled by partially removing the respective macros, or making the lookups unqualified.
- Casts are handled by keeping the cast statement, and calling the AD tool specific API to extract the primal value.
- The single detected (anonymous) union was removed by the AD expert, which is a valid transformation. It is not used in computationally critical path, hence, the additional memory requirements should be negligible overall.
- In a few locations, additional template specializations were added. In particular, an implicit conversion of an integer to a double type had to be resolved. While these conversions are legal in the standard, once the double type is changed to a scalar this can cause a two-step user-defined conversions, as shown in the motivating example, see Section 4.1.

ISSM Table 4.2 shows results of a comparison between (1) a version of ISSM before all AD related transformations were applied to the code base (from 2011) (2) with a more recent one (from 2014), which had all major parts of the AD computations functional. The full results are presented in [63].

As stated in Section 3.1.2, ISSM makes use of templatized cast function, specialised for the AD overloading type, for all conversions in the code.

Reviewing a recent public release version of ISSM (2019), explicit conversion are used more than 210 times in 67 source and header files. Table 4.3 shows the different cast

	Late 2011	Mid 2014
Files	912	857
Lines of Code	66,573	80,044
Translation Units	298	254
#Explicit Conversion in #files	192 in 46	0
#Implicit bool Conversion in #files	44 in 10	0

Table 4.2: ISSM evolution w.r.t. problematic code constructs.

destination types. The last row, in particular, shows casts to the alias for the active type. These are applied, e.g., on passive values to gain activity. Finally, casts to an integer or bool value for the conditions of if statements are found in total 23 times.

	`bool`	`int`	`double`	`IssmPDouble`	`IssmDouble`
Count	27	126	13	32	19

Table 4.3: ISSM type cast counts.

4.5 Discussion

A type change in a (scientific) code can lead to code defects causing an extreme amount of compiler error output, as seen with OpenFOAM. All analysed codes contain issues w.r.t. user-defined types. The exceptions are already AD-augmented codes, which have manually resolved these issues in the past. The required adaptive maintenance can be time-consuming and costly. Unsurprisingly, reports show that a non-trivial amount of work is necessary for a full integration of AD in a complex code base, e.g., OpenFOAM and ISSM. Reviewing the applied code changes for AD, similar changes and solutions are identified.

Overall, this led us to a thorough analysis of code defects and a subsequent identification of the root cause. Section 4.2.1 showed caveats of overloading, and, also, presented solutions to common restrictions of the C++ language, while keeping the impact on the code base to a minimum. The error analysis can thus be used as a blue print for new code development, and helps the assessment of required workload to make a legacy code work with AD overloading or any other overloading-based extension of arithmetic type, e.g., variable

precision arithmetic. To that end, tools to help with the AD workflow are worthwhile as they can significantly reduce the workload for AD. Here, OO-Lint is concerned with finding problematic code constructs that cause compile time errors after a type change. The tool is applied to target codes before the type change. It aims to find any existing problems that would cause errors after the type change if left unchanged. For each match, localized source transformation can be applied.

4.5.1 Limitations of OO-Lint

In this section, the limitations are described before potential solutions are presented.

Template code Detection is hindered by the fact that the template instantiation type does not carry alias information. As a consequence, OO-Lint relies on heuristics to detect if the template type parameter is instantiated by the relevant alias. In Listing 4.17 two template instantiations exist for the same template function. The compiler generates one function instantiation, with the type `double`. The respective AST nodes do not carry the alias information.

```
1  using scalar = double;
2  template<typename T>
3  void foo(T a) {
4    if(a) { ... } // a problematic code construct after a type change
5  }
6  void bar() {
7    // not relevant for type change:
8    foo<double>(double{});
9    // scalar instantiation type known here, but
10   // uses foo<double>, T ≡ double before a type change
11   foo<scalar>(scalar{});
12 }
```

Listing 4.17: The alias information are lost in the instantiation of `foo` — one instantiation is created with the template argument type double. The respective AST nodes are not carrying the required alias information for matching with OO-Lint.

The type string Although the tool is flexible w.r.t. the type string that is checked for, choosing "*double*" lowers the precision. For instance, passing a value to an external library, which is not part of the type change, can not yet be detected, see Listing 4.18.

```
1  extern void foo(double*); // provided by an external library
2  void bar() {
3    double a = 1.0;
4    foo(&a); // breaks after type change
5  }
```

Listing 4.18: After the type change, `a` (line 3) is now an `adouble` and needs to cast before passing it to the external library. OO-Lint does not see a mismatch before the type change.

Resolving these limitations Overall, the aforementioned limitations can be resolved with additional development. As a remedy, OO-Lint detects the instantiations of template code with, e.g., the scalar value (i.e., line 9 in the above code), where the type alias string is still known. In turn, the body of the template instantiation of, e.g., `foo` is analysed by interpreting every AST node w.r.t. the dependent type T[5] as the alias string OO-Lint is configured with, e.g., `scalar`. This is required for template functions and classes where the lost alias information can cause a missed match of a problematic code construct, as shown in Listing 4.17. This heuristic currently works reasonable for simple cases, and will be extended to handle more complicated instantiation patterns.

External libraries can be handled by making OO-Lint aware of, e.g., namespace of external libraries. As a remedy, the tool could analyse the call site, and if it targets some external library function, report it. Hence, the variables from call sites passed to a callee of an external library is interpreted correctly as a mismatch, and flagged as a problematic code location.

4.5.2 Related work

OO-Lint was designed to implement the analysis results and proposed solutions for sources of errors when a user-defined type is introduced. This systematic description of errors pertaining to the introduction of a user-defined type was not done previously. However,

[5] https://en.cppreference.com/w/cpp/language/dependent_name

related to this initial step of a type change is the automatic introduction of the AD overloading type to the target code.

TypeForge [99] is a recent tool that adds AD code annotations, e.g., it replaces floating-point types to a specified alias `AD_real`. The redeclaration is subsequently used in the tool *ADAPT* [96], which uses the alias to introduce CoDiPack to find floating-point precision tuning opportunities based on the sensitivities computed with the RM. TypeForge, though, has not been applied to a code at scale and, also, is not handling any AD-relevant MPI calls yet.

5 TypeART: Type tracking and correctness checking in MPI

In the previous Chapter 4, fundamental problems of the AD overloading approach when applied to codes were discussed. Problems pertained to, e.g., data conversions that need to be made explicit for the code to be correct, otherwise compile time errors occur. In this chapter, runtime type correctness focusing on MPI communication is discussed. Type errors in MPI can be subtle, as the routines work on typeless buffer arrays and the developer is responsible to ensure correct, matching types during transfer.

MPI codes can have millions of lines of code, and contain complicated communication patterns to achieve the highest efficiency. Manually checking these applications for correctness of the MPI communication is not feasible. To reduce this complexity, correctness checking tools like MUST [54] were developed.

In particular, MUST detects, (1) deadlocks, (2) mismatch of buffer sizes, or (3) a mismatch of declared MPI data types in matching communication routines during runtime. MUST operates by intercepting all MPI communication during runtime and analysing these for potential defects. Consequently, the underlying type of the memory passed to an MPI routine as a typeless `void` pointer is unknown to a tool like MUST.

As a remedy, the sanitizer tool TypeART [62] was developed using the LLVM compiler framework to overcome this limitation. TypeART is a type and memory allocation tracking sanitizer. It consists of an LLVM compiler pass and a corresponding runtime to track relevant memory allocation information during the execution of a target program. It instruments heap, stack and global variable allocations with a callback to the TypeART runtime. The callback consists of the memory pointer value, a type id describing the type and the extent of the allocation. At runtime, it provides MUST with an interface to check the underlying type and extent of any memory address passed to an MPI routine. Hence, with TypeART, MUST is not limited to the check of programmer-declared MPI datatypes

but, in addition, can check whether the actual memory location, provided as typeless communication buffer, matches the declared type and extent.

Having the ability to check for type correctness w.r.t. MPI communication is especially important for (1) MPI derived data types, which require error-prone manual construction, and (2) augmenting these target codes with AD, which impacts the existing MPI usage.

Derived data types are used to transfer, e.g., heterogeneous data and require a prior construction and registration with the runtime MPI library. Erroneous memory layout assumptions can cause subtle errors during runtime, see Section 5.1. AD, on the other hand, impacts all MPI communication as the type change of the underlying scalar type requires a change of the transferred types as well. For the FM, instead of sending a single floating-point type, the primal and the derivative value is sent, which doubles the data size. For the RM, typically, the whole MPI interface is exchanged to one of the adjoint MPI implementations, which requires code changes to almost every usage instance of MPI routines. A developer can not easily verify any prior assumptions about the correctness of the AD-augmented code due to these substantial changes. Hence, correctness needs to be reverified by using TypeART with MUST in combination.

The rest of this chapter is structured as follows. Section 5.1 discusses how erroneous data layout assumptions lead to correctness errors. In Section 5.2, the use of MPI with AD is discussed. The MUST approach is briefly introduced in Section 5.3. Subsequently, the TypeART architecture and implementation details are discussed in Section 5.4. First, the TypeART runtime, which tracks type information for allocations, is presented in Section 5.4.1. The static code analysis and instrumentation to extract the required type information are discussed in Section 5.4.2 and Section 5.4.3, respectively. After an evaluation on a set of well-known MPI programs in Section 5.5, a discussion is given in Section 5.6, including avenues of further improvements and use cases.

Reference This chapter is based on the contents of the below listed work. Additional aspects, not previously published, are the discussion and analysis of the impact of the RM on MPI applications, and so-called type asserts, i.e., manually added assert statements to check the type of any pointer to ensure type correctness.

- A. Hück, J.-P. Lehr, S. Kreutzer, J. Protze, C. Terboven, C. Bischof, M. S. Müller. **"Compiler-aided type tracking for correctness checking of MPI applications"**. In: *2nd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, Nov. 2018, pp. 51–58. doi: 10.1109/Correctness.2018.00011.

5.1 Motivating example: Derived datatypes

The involved complications in creating user-defined dataypes for MPI are illustrated in Figure 5.1. It displays two possible memory layouts for how a compiler might arrange a C language `struct` for a specific architecture. The struct consists of one integer and two double precision floating-point variables, i.e., `struct S {int i; double d[2];}`.

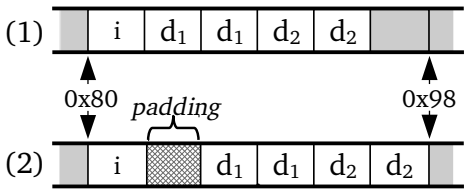


Figure 5.1: Two potential memory layouts of the `struct S` in memory. Each cell of the memory view is a 4 byte word: `int` (`i`) is 4 byte, while a double (d_n) is 8 byte wide. (1) A compact representation with each struct member being subsequently laid out in memory. (2) The compiler adds 4 byte padding to enable an efficient 8 byte aligned memory access. Adapted from [62].

The memory representation of Figure 5.1 (1) is more compact and consumes 4 bytes less per allocation. However, on many architectures, the compilers will generate a memory placement as shown in Figure 5.1 (2), since the enabled aligned memory access typically provides better performance. At the same time, MPI user-defined datatypes for structs are manually constructed by the developer. They are responsible to specify, among other required information, the type and the respective memory offsets to create a *memory overlay* for the MPI library to correctly extract the values (from, e.g., the struct `S`) to transfer correct data. Both memory placements can be expressed with the help of MPI user-defined datatypes, however, not knowing about such particularities, a developer can easily specify a wrong layout. For instance, specifying size offsets only using the built-in size operator[1] for the particular datatype, see Figure 5.2. Unfortunately, such errors do not cause any immediate runtime error. The MPI standard explicitly states in Section 4.1.12 [98]:

> "It is not expected that MPI implementations will be able to detect erroneous, 'out of bound' displacements [...]"

[1] https://en.cppreference.com/w/cpp/language/sizeof

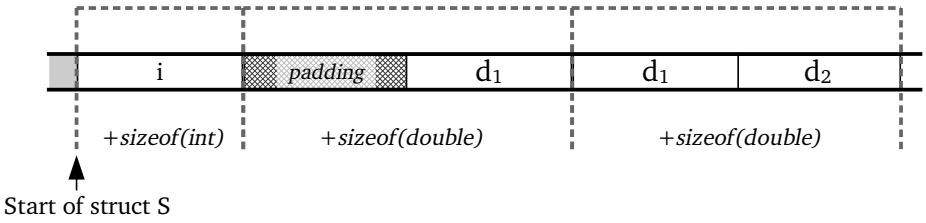


Figure 5.2: To provide MPI with the necessary data member memory position for construction of a derived datatype, the correct offsets from the start address of the struct have to be calculated. Here, using the `sizeof` operator is semantically wrong, because the layout assumption (i.e., Figure 5.1a) of the developer were not met. As a result, the padding is erroneously interpreted as the start of the first double value of the struct. Adapted from [62].

By tracking the memory allocations and respective type layouts with TypeART, this class of type errors is straightforward to detect in MUST, therefore identifying subtle programming errors and portability issues.

5.2 MPI and algorithmic differentiation

MPI support for AD overloading tools is provided by external library implementations of the forward and reverse AD MPI concepts. There exist three main implementations ([119; 121; 131], also called *MPI AD library* henceforth) which are compatible with various, different AD tools.

Common among these are the provision of an abstract interface which the AD tool needs to implement. The interface abstracts away from the concrete implementation of, e.g., accessing the primal and derivative values required for the MPI communication. In particular, for the AdjointMPI library the implementation of the specific interface is shown in Figure 5.3.

The MPI AD libraries take care of the particularities of transferring active types between processes. The MPI feature support is typically incomplete with MeDiPack [119] having, comparatively, the most functionality of MPI implemented for AD.

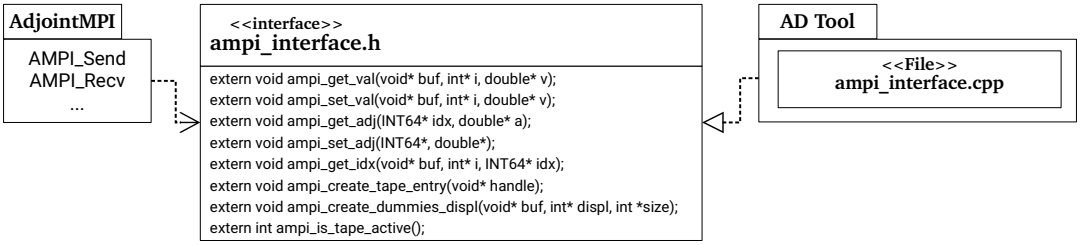


Figure 5.3: AdjointMPI provides an interface which it calls to extract primal values, get indices of the adjoint values and other related bookkeeping operations. Any AD tool needs to implement the interface and link with the library. Adapted from [59].

RM and MPI The aforementioned libraries are primarily concerned with the correct implementation of MPI and RM. As shown in Section 2.1.1, the RM requires a complete reversal of the data flow, which also holds for the MPI communication related to active types. Each MPI process has its own private tape, which traces the data flow of that process only. Whenever AD-related MPI communication between processes occurs, the correct adjoint propagation are dependent on the other processes it communicated with. Hence, if process 0 sends active data a_0 to process 1, for the correct adjoint propagation, process 1 needs to send back the relevant adjoints to process 0 during the reverse evaluation, see Figure 5.4. The above pattern w.r.t. the reversal of `MPI_Send` and `MPI_Recv` communication pairs has to be extended to other, more complex communication patterns found in HPC codes, see [121] for more details.

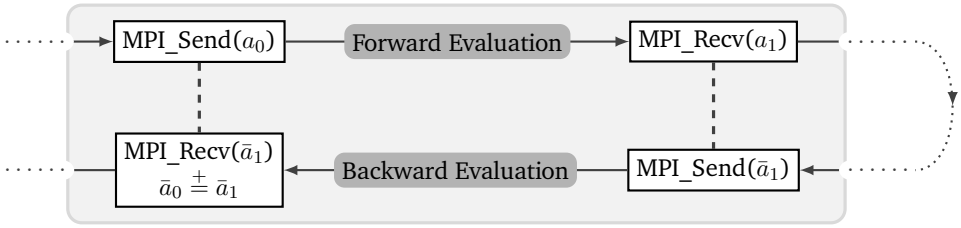


Figure 5.4: Execution excerpt of an adjoint MPI program. During *forward evaluation*: The rank 0 process sends data a_0 to some other process. During *backward evaluation*: To update the adjoints of rank 0 requires a reversal of the communication, the other process now sends the respective adjoint values $\bar{a}_1$ to rank 0 for the update.

The communication routines of the MPI AD libraries only transfer the primal values of the AD RM overloading type, see Figure A.3 for an illustration of such an implementation. For the required reversal, the external function interface of the AD tool is used, i.e., whenever a relevant communication is handled, an entry on the tape is created which indicates the execution of the reversed communication pattern.

FM and MPI The FM, in contrast to RM, does not require a reversal of the MPI communication patterns, and hence can be implemented with either (1) duplicating the MPI communication (called *shadowing* [128]), therefore, sending the primal values and derivative values separately with the same communication mode, or (2) by, e.g., doubling the size of the buffer and sending both the primal and derivative values as a contiguous chunk of memory, where the primal and derivative value are stored as subsequent pairs [130].

MPI derived datatypes Consider the struct shown in Figure 5.1 with active types (e.g., `adouble`) replacing the double type members, see Figure 5.5. Transferring this struct as a derived data type with MPI requires the extraction of the primal values of these active members in the context of the RM. Handling this correctly is the responsibility of the MPI AD library by providing overloads to the necessary MPI derived type constructors which contain additional book-keeping code for packing and unpacking [131].

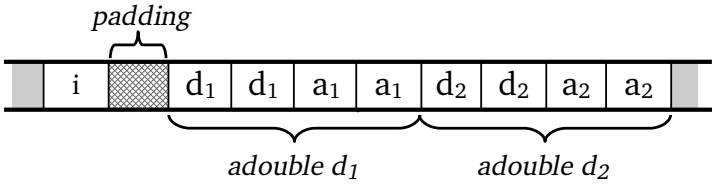


Figure 5.5: The struct layout of Figure 5.1 with some AD overloading type which has, as data members, the primal value and an address pointer ($\mathtt{a}_i$) to the tape location for its adjoint value. As a result, the size of the struct is 40 bytes.

In the context of MPI type correctness, AD adds another layer of complexity that, fortunately, is mostly handled by the respective MPI AD library implementation. However, the data layout assumptions can fundamentally change with the introduced AD type, as shown in Figure 5.5. This especially necessitates the correct type registration and communication of complex types with MPI.

5.3 Runtime analysis of MPI datatypes with MUST

Matching the static type with the type of the passed typeless buffer is done inside the MUST tool. The standard defines MPI basic datatypes as well as the aforementioned user-defined types, which require manual construction by the developer with a predefined API. In either case, the MPI standard [98] describes the following type matching rules for the transfer of data in Section 3.3.1:

> "One can think of message transfer as consisting of the following three phases.
>
> 1. Data is pulled out of the send buffer and a message is assembled.
> 2. A message is transferred from sender to receiver.
> 3. Data is pulled from the incoming message and disassembled into the receive buffer.
>
> Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous."

MUST, prior to the TypeART integration, observes information of MPI communication by wrapping the calls. With the available MPI function call arguments, MUST's type checking is limited to the base address of the buffer, the declared MPI datatype and the count. As such, the information provided by wrapping MPI calls limits type checks in MUST to phase two in the above-described three phases of message transfers. With TypeART the full phase of the MPI data transfer can be checked. MUST's extended type checking with TypeART is further described in [62].

5.4 TypeART design and implementation

TypeART is a sanitizer tool for runtime correctness checks between the declared MPI datatype and the dynamic datatype of the underlying buffer used in the MPI communication routines. It is based on the LLVM 6 compiler toolchain and consists of an LLVM

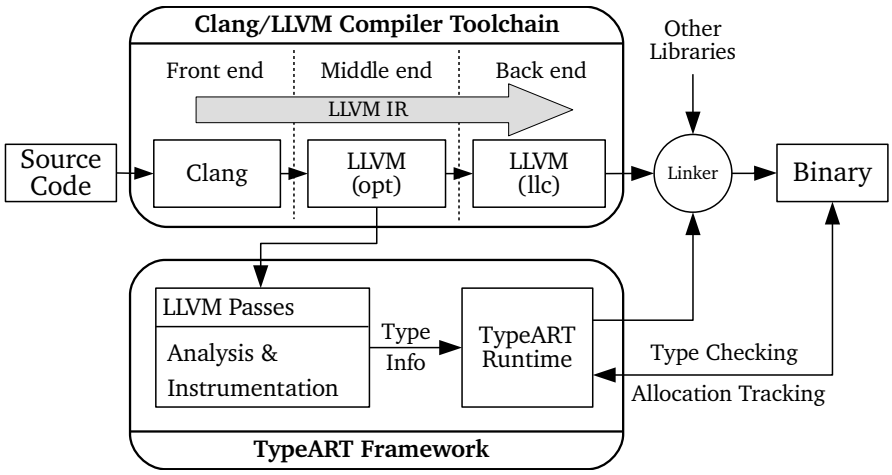


Figure 5.6: TypeART work flow, adapted from [62]. (1) The mid-end (`opt`) is invoked with additional analysis and transformation passes to instrument every relevant memory allocation, and collect type information during compilation. (2) The TypeART runtime is linked into the target MPI application. (3) During execution, it traces the invoked and instrumented memory allocations.

compiler analysis and instrumentation pass and a runtime library. The overall design and work flow of TypeART is shown in Figure 5.6.

To achieve the goal to enable type correctness checks for any MPI buffer, all relevant memory allocations in a target application need to be instrumented. To that end, the runtime defines a C language API for the instrumentation callbacks and, in addition, provides an API to query type information of a memory address. Additionally, TypeART offers so called type assert functionality. These assert statements can be added to a code to verify assumptions about the type behind a (typeless) pointer, and, if the assumption does not hold the program, e.g., terminates with the standard C language asserts.[2]

5.4.1 TypeART runtime

The runtime component keeps track of the different memory locations at program runtime per MPI process. To provide detailed type information, e.g., what struct and which offsets

[2] https://en.cppreference.com/w/cpp/error/assert

between the struct members are required, it also defines an API for, e.g., MUST to query this information.

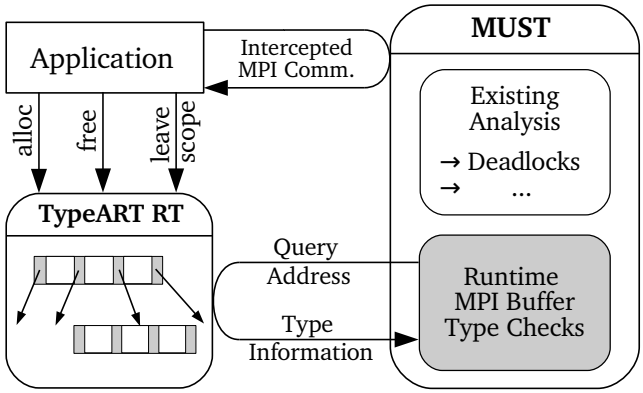


Figure 5.7: TypeART runtime and MUST interaction with a target MPI program. The instrumented MPI application calls the TypeART runtime whenever (1) a heap, stack or global variable allocation occurs, (2) a free on a heap allocation is called, or, (3) a scope is left (e.g., returning from a function call) and tracked stack variables are deallocated. MUST calls the runtime whenever an intercepted MPI call requires a type check, the returned type information are matched with the static MPI datatype. Adapted from [62].

In Figure 5.7, we show how the runtime interacts with the target MPI program through the inserted instrumentation calls and how MUST interacts with both to analyse the MPI communication with the existing analysis, e.g., deadlock detection, and our newly implemented type checking capabilities.

The allocated memory locations are, internally, stored with a btree-based ordered map implementation. The key for the map is the memory address and the value is a *pointer info* struct, storing (1) a type id, identifying each unique type in a program, (2) a count for the size of, e.g., an array, and (3) optionally, the memory address where the allocation occurred for diagnostics. Hence, the memory size required to track each allocation is between 16 and 24 bytes.

Whenever a heap or stack deallocation occurs, the pointer is removed from the map to ensure that no stale metadata is kept. While the btree map is used as a global storage for tracked stack, heap and global memory addresses, the stack requires an additional, internal data structure. It keeps track of the active stack memory addresses of the current

scope, and is used for our counter-based deallocation scheme to free all stack addresses of that scope.

5.4.2 Analysis pass

The analysis pass (1) searches the LLVM Intermediate Representation (IR) and collects all memory allocation instructions, and (2) filters unwanted stack and global variable allocations that are provably not passed to any MPI routine as a buffer address.

Heap memory allocation and deallocation Heap memory allocations are represented in the IR as call instructions to, e.g., the C-function `malloc`. The type of the pointer returned by the malloc is only determined after a subsequent typecast operation. In Listing 5.1, an example of a C heap memory allocation and the corresponding IR code is shown.

```
1 (float*) malloc(sizeof(float));
2
```

```
%1 = call i8* @malloc(i64 4)
%2 = bitcast i8* %1 to float*
```

Listing 5.1: Typical IR pattern of a heap memory allocation. C code and corresponding IR for a `malloc` call and a dependent `bitcast` of the memory pointer.

In complex IR codes, the pointer returned by the heap allocation might be cast multiple times to different types. The pass walks the LLVM-provided use-def chain of each heap allocation instruction to collect all bitcasts, but typically the first one to a non byte-like pointer type is sufficient to determine the dynamic type. Finally, deallocation instructions (e.g., `free` in C) are collected, and no further analysis is required here as only the pointer value is relevant for the runtime address management.

Stack memory allocations Stack allocations are represented as `alloca` instructions. The type of the allocation is directly encoded. Thus, we do not need to search for type specifying instructions, see Listing 5.2.

Global variables Global variables are declared per TU in the IR (called module), and can also be introduced by LLVM for internal usage. Thus, per TU, all global variables, e.g., LLVM-specific intrinsics, or are declared but not defined in the current TU are filtered out. The type is determined as with stack allocations, see Listing 5.3.

```
1 float a[2];
2 int i;
```

```
%1 = alloca [2 x float], align 4
%2 = alloca i32, align 4
```

Listing 5.2: Stack variables in C and corresponding instructions in the IR. The type can be queried directly on the allocation.

```
1 float a[2];
```

```
@a = common global [2 x float] zeroinitializer, align 4
```

Listing 5.3: Global variable in C, in the IR these are stored on a module level.

Filtering stack and global allocations The overhead of tracking stack allocations is potentially significant for complex programs. At the same time, many of these stack allocations will never be part of the MPI communication. The same assumption holds for global variables. Clearly, filtering out these unwanted allocations will improve performance.

Hence, the analysis employs a conservative forward data flow analysis: It tracks the usage of the stack allocation throughout the enclosing parent function, as well as the use of global variables. If the allocation is passed to a function call, the analysis checks if (1) the callee is an MPI routine, and, additionally, if the allocation is passed as a buffer object. In this case, the allocation is relevant for instrumentation and, therefore, not filtered. (2) Alternatively, the callee is further analysed with an argument position dependent data flow analysis following the call graph. Here, indirect calls, where the definition is not available, cause the filter operation to pessimistically assume that the stack object is relevant for the later instrumentation step. Likewise, if the argument position dependence can not be resolved for the callee analysis, the whole sub-call graph must be free from any MPI call to allow filtering the allocation.

In Listing 5.4, a simple C example is shown for a successful filter analysis and a failed one due to an indirect call. Consequently, the filtering opportunities are restricted to the function definitions of the currently analysed TU, as calls to functions defined in other files are typically indirect calls.

5.4.3 Transformation pass

The transformation pass uses the preprocessed collection of allocation instructions of the analysis pass and adds instrumentation hooks for the runtime. The instrumentation of allocation instructions works as follows: (1) The TypeART *type id* of the IR allocation type

```
1   extern foo_bar(int*); // The definition is not available
2   void bar(int* x, int* y) {
3     *x = 4;             // x is not used after the assignment
4     MPI_Isend(y, ...);  // y is passed to an MPI routine
5   }
6   void foo() {
7     int a = 1, b = 2, c = 3;
8     bar(&a, &b);
9     foo_bar(&c);
10  }
```

Listing 5.4: Three relevant cases: (1) The filter follows the data flow of `a`, and reaches the function call to `bar`. The allocation is passed as the first argument, as such the analysis continues with the data flow of the first parameter `x` of the callee `bar`. The analysis detects a filtering opportunity for `a` as the data flow ends, and it is not part of an MPI call. (2) For `b`, the aliasing pointer `y`, on the other hand, is used in an MPI call and can not be filtered. (3) Likewise, `c` can not be filtered, as it is passed to an indirect function call.

is generated for runtime type identification, (2) the number of allocated elements is determined (using the argument passed to, e.g., `malloc`), and finally, (3) an instrumentation call is inserted into the IR, passing this information to the runtime. For globals, so-called constructor functions are introduced to a module which is run at program startup, similar to [86]. For instrumentation calls of heap deallocation functions, e.g., `free`, only the pointer address is required to remove it from the runtime tracking.

Type representation with a type id In order for the runtime to identify and handle built-in and user-defined types, the transformation pass generates a unique ID for each type encountered with an allocation instruction. Each time a new user-defined type is found, (1) its layout is serialized, and (2) a unique type id is generated for the runtime to use for the type-aware layout matching. For built-in types, the type id and size is predetermined. Pointers to any type of, e.g., a struct member variable, have a special type id to indicate MUST that it needs to recursively handle the type behind the pointer. In Listing 5.5 an example of a user-defined struct with multiple members is shown.

```
1  struct s1_t {
2    char a[3];
3    struct s2_t* b;
4  };
5
6
7
8
9
```

```
- id:              256
  name:            struct.s1_t
  extent:          16
  member_count:    2
  offsets:         [ 0, 8 ]
  types:
    - {id:   0, kind:  builtin}
    - {id:  10, kind:  pointer}
  sizes:           [ 3, 1 ]
```

Listing 5.5: Serialized type information of struct `s1_t` on the right (yaml format). Type ids below 256 are reserved for built-in types. The extent of the struct is given in bytes. For each member of the struct: Byte offsets from the base address of `s1_t` and the respective sizes are listed.

The type ids are passed with the instrumentation hooks to the runtime. The runtime reads the file containing the serialized type information of a target project to a database and, thus, has exact type information for each instrumented allocation. Type information for member pointers, as shown in Listing 5.5 (`struct s2_t*`), are resolved by querying the runtime using the pointer memory address: Either a separate tracked allocation occurred for this member, or the pointer was never initialized. This has to be recursively repeated for each such instance.

Heap allocated variables The instrumentation of heap related memory functions for C and C++ works similarly. In Listing 5.6 an example for a call to `malloc` is shown. For `calloc`, the calculation of the number of types is omitted as it is explicitly passed as a parameter and can, thus, be extracted from the call arguments itself.

```
1  d_ptr = (double*) realloc(d_ptr, 20 * sizeof(double));
```

```
1  %1 = load double*, double** %0, align 8 ; load the d_ptr
2  %2 = bitcast double* %1 to i8*
3  call void @__typeart_free(i8* %2)
4  %3 = call i8* @realloc(i8* %2, i64 160)
5  call void @__typeart_alloc(i8* %3, i32 6, i64 20)
6  %4 = bitcast i8* %3 to double*
```

Listing 5.7: Instrumented IR code for a C `realloc` call. The address passed to the realloc call is first freed in the runtime. Subsequently, the returned address of the reallocation is registered with the runtime.

```
1 (float*) malloc(n * sizeof(float));
```

```
1 %1 = call i8* @malloc(i64 %0)   ; %0 = n * sizeof(float)
2 %2 = udiv i64 %0, 4             ; %2 = %0 / sizeof(float) = n
3 call void @__typeart_alloc(i8* %1, i32 5, i64 %2)
4 %3 = bitcast i8* %1 to float*
```

Listing 5.6: Instrumented IR code for a C `malloc` call. The lines with the gray background were added by the transformation. In line two, the number of types that were allocated is computed. The instrumentation hook then passes to the runtime (1) the pointer address returned by malloc, (2) the unique type id (here 5 for `float`), and, finally, (3) the number of float elements (`n`).

A special case is the `realloc` call, see Listing 5.7, which either returns the same pointer with a new extent of memory (expanded or contracted), or returns a new pointer with the original values copied over. Thus, the transformation first adds a runtime call to free the pointer passed to `realloc` and, after the call, the returned address is passed as a new allocation to the runtime (as an instrumented call similar to, e.g., `malloc`).

Stack allocated variables Stack variables have automatic lifetime properties. If the relevant scope is left, the stack variables of that scope are automatically freed. Typically, a function enter and exit instrumentation with hooks can be done to handle the correct scope specific stack handling. However, this approach can induce a significant additional overhead, thus, the costly function enter instrumentation is omitted by using a counter-based approach: (1) A counter is introduced per function scope and initialized to zero. (2) Each stack allocation is instrumented, similarly to the heap allocations. (3) The total amount of stack allocations that occurred in each basic block of the function is added to the counter. In the IR a block is a set of instructions with a single entry and exit section (i.e., no control flow in-between). (4) For each function exit, the counter value at that point is passed to the runtime for the correct, internal stack clean up operation. In Listing 5.8, a simplified example, without control flow, and one function exit, is shown.

```
1  void foo() {
2    float a[2];
3    int i, j, k;
4  }
```

```
1  define void @foo() {
2   %__ta_alloca_counter = alloca i64
3    store i64 0, i64* %__ta_alloca_counter
4    %1 = alloca [2 x float], align 4
5    %2 = bitcast [2 x float]* %1 to i8*
6    call void @__typeart_alloc_stack(i8* %2, i32 5, i64 2)
7    ; alloca for int i, j, k not shown, similar to float a[3].
8    %9 = load i64, i64* %__ta_alloca_counter
9    %10 = add i64 4, %9
10   store i64 %10, i64* %__ta_alloca_counter
11   call void @__typeart_leave_scope(i64 %__ta_alloca_counter)
12   ret void
```

Listing 5.8: The C function has 4 stack allocations and no additional scoping. The allocation handling is specific to the scope: A counter is defined and initialized to 0 (line 2–3) which is subsequently used to count the number of stack allocations per scope. The runtime is called for each stack allocation (line 5–6). The counter is, at the end of the scope, incremented by 4 (line 8–9). Once the scope ends, the total count is passed to the runtime for the internal stack cleanup operation (line 11).

Global variables The instrumentation of global variables is done by (1) introducing a *constructor* function using the LLVM API[3], which is run at program start up, to the current TU, and (2) adding an instrumentation call to our runtime for each global variable inside the constructor functions body, similar to heap and stack allocations. De-registering global variables is skipped as the lifetime of them is tied to the program execution lifetime. In Listing 5.9 the instrumentation of a global float variable is shown.

Type assertion Asserts are used to verify assumptions about the type of pointer values. They are manually added to the code, likewise to the standard asserts of the C language. The asserts are implemented as skeleton macros, see Listing 5.10, which are replaced by the instrumentation pass with appropriate API calls to the runtime, see Listing 5.11

[3] https://llvm.org/docs/LangRef.html#the-llvm-global-ctors-global-variable

```
float global_var;
```

```
@global_var = common global float 0.000000e+00, align 4
@llvm.global_ctors = appending global
    [... { i32 0, void ()* @__typeart_init_module, i8* null }]
define private void @__typeart_init_module() {
entry:
  %0 = i8* bitcast (float* @global_var to i8*)
  call void @__typeart_alloc_global(i8* %0, i32 5, i64 1)
  ret void
}
```

Listing 5.9: The instrumentation adds a constructor function to each module if it contains a global. The constructor is called at startup. The global is registered with the runtime in the introduced constructor (line 4–9). Using the LLVM intrinsic global array variable `llvm.global_ctors`, the function is registered with additional arguments (skipped for brevity) (line 2–3).

```
#define ASSERT_TYPE(ptr, type, len)                \
   { type* __type_ptr;                             \
     __typeart_assert_type_stub(ptr, __type_ptr, len); }
void foo(void* x) {
  ASSERT_TYPE(x, double, 2) // void* x must be type double with extent 2
}
```

Listing 5.10: A type assert macro implementation and usage. It adds a pointer declaration and a no-op function.

5.5 Evaluation

The empirical study of TypeART focuses on (1) coverage and correctness of the static instrumentation pass, (2) stand-alone runtime and memory overheads induced by the type tracing, and, finally, (3) overall impact of TypeART when fully integrated into MUST, including MPI type checks.

First, a synthetic benchmark is discussed in Section 5.5.1. It presents empirical runtime results of common operations of the TypeART runtime, due to (1) the added instrumentation hooks to track de-/allocations of memory, and (2) type related queries done by a tool like MUST. This is mostly a benchmark of the underlying map tracking the memory

```
1 define void @foo(i8*) #0 {
2   %1 = alloca i8*, align 8
3   store i8* %0, i8** %1, align 8
4   %2 = load i8*, i8** %1, align 8
5   call void @__typeart_assert_type(i8* %2, i32 6, i64 2)
6   ret void
7 }
```

Listing 5.11: The transformation replaces these statements with an assert call to the runtime: (1) The pointer to be checked is the first argument. (2) The second argument is the type id of the expected type, which is resolved by using the `type*` declaration. (3) The final argument is the (optional) number of expected elements.

addresses, though. Subsequently, in Section 5.5.2, the impact of TypeART alone and MUST, with and without TypeART integration, on a set of MPI applications is presented.

All benchmarks were run on compute nodes of the Lichtenberg high-performance computer of TU Darmstadt, with two Intel Xeon Processor E-2670 at a fixed frequency of 2.6 GHz and 32 GB RAM. To compile the benchmarks, the Clang compiler $6.0.0$ with Open MPI $3.1.1$ was used.

5.5.1 Synthetic benchmarks

This section empirically quantifies overheads of the frequently executed operations on the runtime with several micro benchmarks. For each such benchmark, the underlying memory map of the runtime is filled based on a randomized vector with address values in the range $[0,n]$. The average time of 300 repetitions for each data point is used.

Memory tracking API

Operations added by the instrumentation pass to a target MPI application are simulated. The measurements, see Figure 5.8, are w.r.t. inserting and erasing both stack and heap memory pointers, respectively.

Insert heap or stack Inserting heap and stack data is done similarly. Each of the n values of the vector is inserted into the runtime and the elapsed time is reported.

Erase heap or stack Erase operations remove existing values in the runtime by simulating a free for heaps and a closing scope for stack memory, respectively. First, the runtime is primed with n values of the vector, and, subsequently, the vector is randomized again for the next timed phase. The runtime-tracked values are removed in random order based on the shuffled vector values, and the total elapsed time is reported.

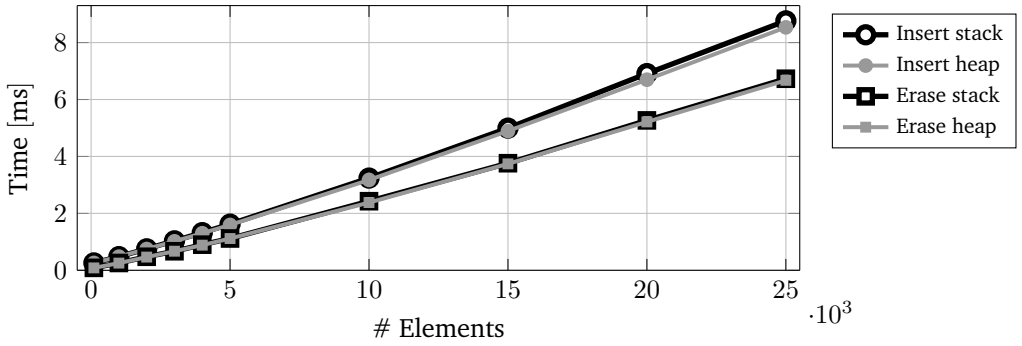


Figure 5.8: Stack and heap operations on the runtime.

The results shown in Figure 5.8 overlap as the cost is almost identical. For the stack related operations, with a rising number of elements tracked in the runtime, the higher cost of having an additional vector to handle the scoping rules of stack allocations becomes apparent. Freeing a stack variable, while having a higher cost than the freeing a heap address, is only slightly more expensive.

Type query API

Querying data behind a memory pointer and resolving the underlying type information are measured, see Figure 5.9. Here, the runtime is primed with n elements of the vector. The query is done, randomly, based on 10% of the vector elements for each case.

Query (missing) pointer The query of a pointer address checks if the runtime contains type information for some address. Hence, a query for non-existent addresses always fails, and returns no such type information.

Resolve type More type information behind an address are extracted compared to the pointer query. For instance, the extent and base address are returned for each call and is, as such, a more expensive operation.

The results in Figure 5.9 show that query operations are relatively cheap overall. A query of a non-tracked value is cheaper since there is no additional copy operation of information,

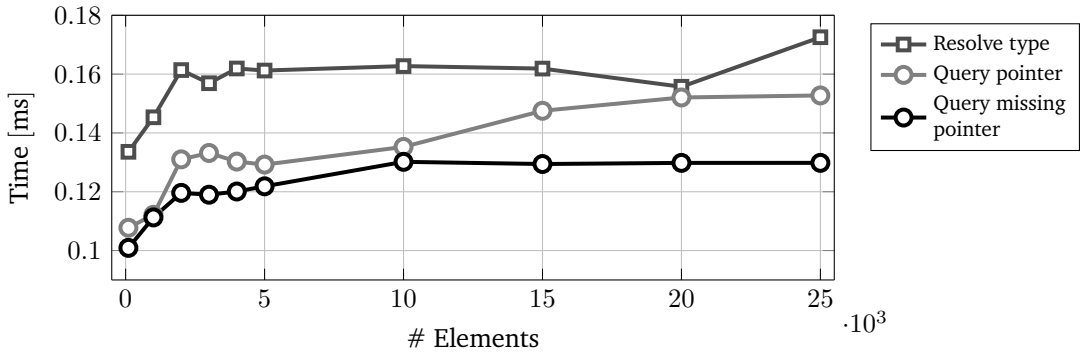


Figure 5.9: Querying and resolving type information with the runtime.

which happens when an existing value is queried. Resolving detailed type information of a memory address has the highest cost, as additional values such as the base address are computed and returned.

5.5.2 MPI applications

The evaluation is based on two applications of the CORAL benchmark suite [24] and two applications of SPEC MPI 2007 [101], respectively. Any parallelism support other than MPI was disabled for each benchmark. The default optimization flags provided by the respective build systems is used (`-O2` or higher). In addition, debug information (`-g`) were included for all runs as this is a necessary configuration for the MUST tool to generate useful diagnostics including source code references. The CORAL and SPEC MPI benchmarks are executed with 8 and 16 processes on two compute nodes, respectively. An additional process is reserved for MUST, as it uses a separate MPI process for global analysis. Hereafter, *vanilla* refers to the unmodified benchmark. The benchmark results of the selected five MPI applications are the median over five runs. The standard deviation of the results w.r.t. media is less than 5%.

CORAL LULESH 2.0 A shock hydrodynamics application that is used as a proxy application for different studies. It performs a stencil operation using MPI for parallelism. The parameter `s`, indicating the number of elements of the cube mesh of one side in a single domain, is set to 38, see [94]. All other parameters are defaults.

104.milc A physics quantum chromodynamics solver. For the presented benchmarks, the data set `mref` was used.

CORAL amg2013 A parallel multigrid solver is employed in the context of unstructured grids. The heap allocation calls of the code base were modified, as the code makes use of type-agnostic allocation wrappers with a set of macro statements for typecasting, see Listing 5.12. This caused the pass to fail to determine the type of the allocated memory. The modifications remove the untyped wrappers and, instead, invoke the C memory allocation functions directly inside the respective macros in a semantically equivalent way, see Listing 5.13.

```
1   char* hypre_MAlloc(int size) {
2      char *ptr;
3      if (size > 0) {
4         ptr = malloc(size);
5      } else {
6         ptr = NULL;
7      }
8      return ptr;
9   }
10  #define hypre_TAlloc(type, count) \
11    ( (type*) hypre_MAlloc((unsigned)(sizeof(type) * (count))) )
```

Listing 5.12: Malloc allocation wrapper. The macro (line 10–11) is used for the heap allocation and is unsupported by the analysis.

```
1   #define hypre_TAlloc(type, count)                              \
2     ( (unsigned)(sizeof(type) * (count)) ) > 0 ?                 \
3         ( (type*) malloc((unsigned)(sizeof(type) * (count))) ) \
4         : (type*) NULL
```

Listing 5.13: Modified macro. The allocation wrapper is removed.

122.tachyon A parallel ray tracing application. The data set `mtest` is used to keep the runtime manageable. With `mref` the simulation takes over 20 minutes for vanilla.

Static coverage

In Table 5.1, the statically collected information of the instrumentation pass is shown, including counts for memory instructions and user-defined types. The amg2013 code base has the most detected unique heap allocations of any code at over 1,400 instrumented statements. As expected, the codes contain many stack allocation instructions. The

amount of stack allocations was partially filtered due to the data flow analysis identifying allocations not part of any MPI call. Filtering is most effective for global variables where the symbol is available for analysis. User-defined types denotes types that are not built-ins, such as structs and other such constructs which were identified by the type identification analysis.

	Memory Operations				User-def.
	Heap	Free	Stack [%]	Global [%]	Types
LULESH 2.0	14	6	54 [21.0]	80 [100]	10
amg2013	1,491	1,152	958 [40.7]	653 [99.4]	61
104.milc	91	64	207 [21.3]	736 [95.4]	25
122.tachyon	80	51	579 [2.0]	372 [97.3]	50

Table 5.1: Static instrumentation and filtering statistics. Stack and Global numbers represent an unfiltered count. The filter percentage is shown in brackets [%].

Dynamic coverage

In contrast to the static values, Table 5.2 shows the median of memory allocation trace information across the MPI processes and the actual amount of required MPI type check operations over all MPI processes by MUST. Here, `Max Stored Heap` and `Max Stack Depth` refer to the maximum amount of concurrently held address pointers in the runtime for heap and for stack allocations, respectively. The global variables tracked by the runtime were never part of any MPI communication, except for 104.milc, which uses one in an MPI broadcast operation. All queried address values by the MPI type checks were tracked by the runtime, there was no missed check. Hence, the static detection found all relevant allocations and the filter did not falsely remove any of them.

The amg2013 benchmark contains the most heap allocations during runtime at over 27 million, and as shown previously, also has the most static heap allocation statements. However, only few of these addresses are ever checked in MUST. In contrast, 122.tachyon has the most stack operations at a total of over 78 million and a maximum stack depth of 277 concurrently held stack address information. Here, all MPI communication also uses unique allocations as indicated by the 482 MPI type checks by MUST on the same number of unique memory addresses. With LULESH 2.0, MUST checks the most addresses at over 40,000, as it has the most amount of MPI communication. However, the unique buffer addresses that were checked are limited to 16 in total, which shows that the buffers are constantly reused for communication.

	Traced Memory Operations					MPI Type Checks	
	Total Global	Total Heap	Total Stack	Max Stored Heap	Max Stack Depth	Total	Unique Checked
LULESH 2.0	0	525,060	34,149	76	21	40,694	16
amg2013	1	27,587,586	2,943	20,736,474	80	1,906	542
104.milc	34	41,638	5,876	79	26	9,206	84
122.tachyon	10	13,759	78,307,707	13,677	277	482	482

Table 5.2: Runtime collected values for (1) Traced memory operations, and (2) MPI-related type checks. The median of all process values is shown. Standard deviation w.r.t. the median is below 2%. An exception are the unique checked addresses. Notably, 122.tachyon has a deviation of about 350% due to one process with 7201 type checks. The others range from 13% to 48%.

Runtime and memory overheads

Finally, the runtime and memory overheads w.r.t. vanilla of TypeART, MUST and MUST with TypeART are shown for each application.

Runtime In Figure 5.10, the relative runtime overhead regarding the vanilla configurations for each benchmark is shown. The runtime overhead is below a factor of 1.5 for all benchmarks, except 122.tachyon. Here, the TypeART runtime and subsequently MUST with TypeART add a runtime overhead factor of over 3. This is explained by the high amount of observed total stack allocations for the tested configuration.

Memory In Figure 5.11, the relative memory overheads regarding the vanilla configurations are shown for each benchmark. Memory consumption is computed as the median over all MPI processes using the maximum resident set size (RSS) measured at the point where `MPI_Finalize` is invoked. Memory overhead is below 1.2 for all benchmarks. For amg2013, the most memory overhead is exhibited as concurrently over 20 million heap addresses are stored at one point during the execution, see Table 5.2.

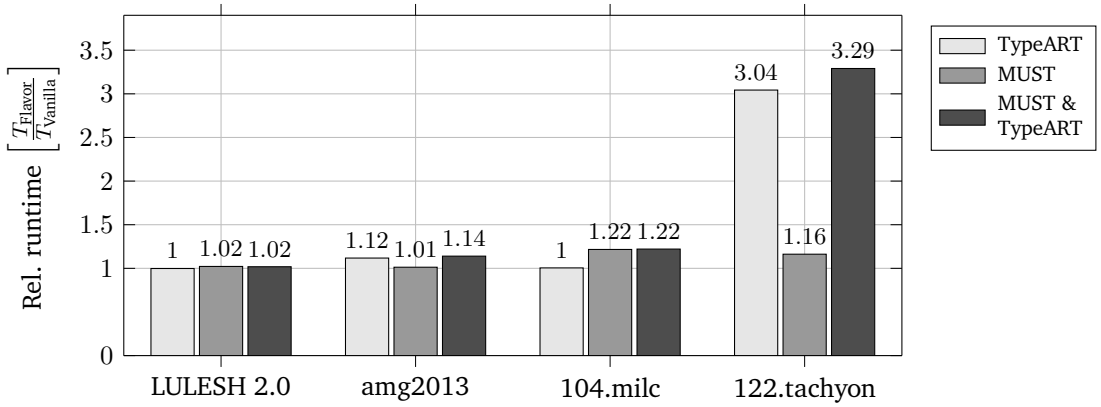


Figure 5.10: The total runtime overhead for the different benchmarks w.r.t. vanilla. Vanilla runtime: (1) LULESH 2.0: 211.36 s, (2) amg2013: 108.96 s, (3) 104.milc: 405.1 s, (4) 122.tachyon: 8.82 s.

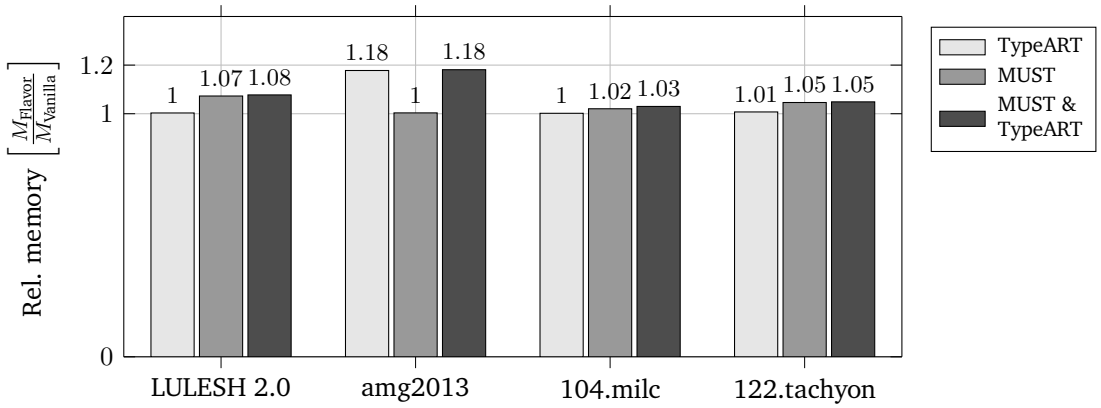


Figure 5.11: The relative median memory overhead of a single MPI process for the different benchmarks w.r.t. vanilla. Vanilla RSS: (1) LULESH 2.0: 140 MB, (2) amg2013: 3,235 MB, (3) 104.milc: 742 MB, (4) 122.tachyon: 265 MB.

5.5.3 Adjoint MPI Application

In the previous section, several primal MPI applications were checked. In this section, an adjoint implementation of LULESH 2.0 is evaluated and compared with the primal variant. The configuration was changed to keep the AD induced overhead manageable, i.e., `s` is set

to 20 and the iterations are capped at 500. Due to software changes on the Lichtenberg HPC system, Open MPI 3.1.4 was used.

AD-enhancement We applied a type change: The underlying floating-point type was replaced with the CoDiPack [118] RM overloading type. The MPI communication was replaced with the adjoint MPI library MeDiPack [119]. For each time-step, in a blackbox fashion, the derivative of the energy (e) at the origin of the domain w.r.t. the pressure (p) is computed. The tape is reset after each time-step. The report produced by LULESH regarding, e.g., the calculated error values agree up to round-off compared to the primal. In Section A.3, aspects of the type change are further highlighted.

Coverage

Static coverage In Table 5.3, the information of the instrumentation is shown.

	Memory Operations				User-def.
	Heap	Free	Stack [%]	Global [%]	Types
AD LULESH 2.0	355	536	1924 [68.7]	878 [83.0]	205

Table 5.3: Static instrumentation and filtering statistics. Stack and global numbers represent an unfiltered count. The filter percentage is shown in brackets [%].

The overall increased count of instrumented memory-related operations can be explained by the header-based implementation of CoDiPack and the majority of MeDiPack, which causes the inlining of these library codes. Hence, a high amount of internal operations are detected by the TypART pass and, subsequently, instrumented. The high number of user-defined types is also explained by these reasons. MeDiPack uses many (adjoint) MPI related data structures. Likewise, CoDiPack exposes several internal structures (e.g., of the underlying tape) that are extracted by the type identification system.

Dynamic coverage In Table 5.4, the runtime collected statistics of TypeART are shown for both the primal and the AD-enhanced code. The AD variant has a significantly higher number of tracked variables. Over 20 million stack variables are tracked overall during the execution, even though the maximum stack depth is only 32. This can be explained by the expression templates (and the inlining) which may introduce more stack variables that are subsequently tracked but also discarded regularly. The heap allocations are partially

explained by the additional memory management for the internal buffers of the adjoint MPI library. The temporary buffers are allocated to transfer the primal data values. Hence, the unique address checks are affected — going up from 16 for the primal to almost 700. The total number of type checks seems consistent between the primal and the adjoint code. In total, 500 time steps were executed. For each step, an MPI reduction is executed to calculate the time increment. The 7,000 additional calls tracked for the adjoint code can be attributed to the required reversal of communication.

	Traced Memory Operations					MPI Type Checks	
	Total Global	Total Heap	Total Stack	Max Stored Heap	Max Stack Depth	Total	Unique Checked
AD LULESH 2.0	149	177,132	24,071,027	215	32	14,506	666
LULESH 2.0	0	100,060	6,524	76	21	7,506	16

Table 5.4: Runtime collected values for (1) Traced memory operations, and (2) MPI-related type checks. The median of all process values is shown. Standard deviation w.r.t. the median of the adjoint variant is below 2% except for the unique checked addresses with about 23% (13% for the primal).

Runtime and memory overheads

Runtime The relative runtime overhead is illustrated in Figure 5.12. Due to the smaller test case configuration, the overhead of MUST is more pronounced, especially with TypeART enabled, compared to the evaluation of the previous section. In contrast, TypeART itself induces only little overhead.

Memory The relative memory overheads are shown in Figure 5.13. The memory overhead factor of AD vanilla compared to the primal vanilla is 1.23. TypeART's induced overhead is negligible with about 4 MB for the AD variant and 1.5 MB for the primal. MUST combined with TypeART for both target code variants adds a fixed overhead of approximately 20 MB compared to vanilla.

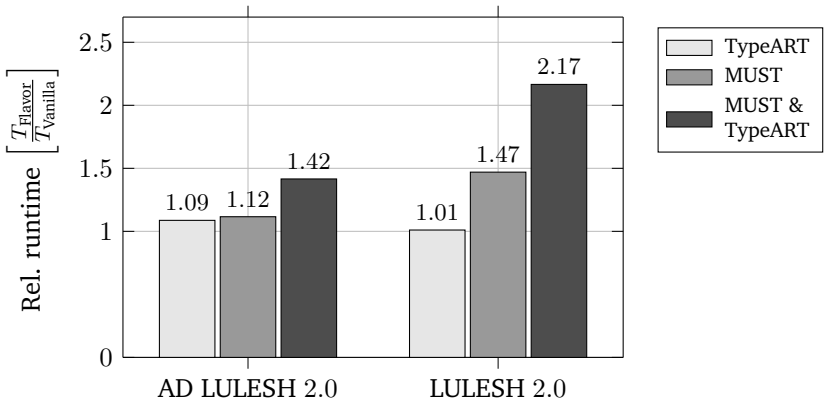


Figure 5.12: Runtime overhead w.r.t. vanilla. Vanilla AD runtime: 93.43 s. Vanilla primal runtime: 7.6 s.

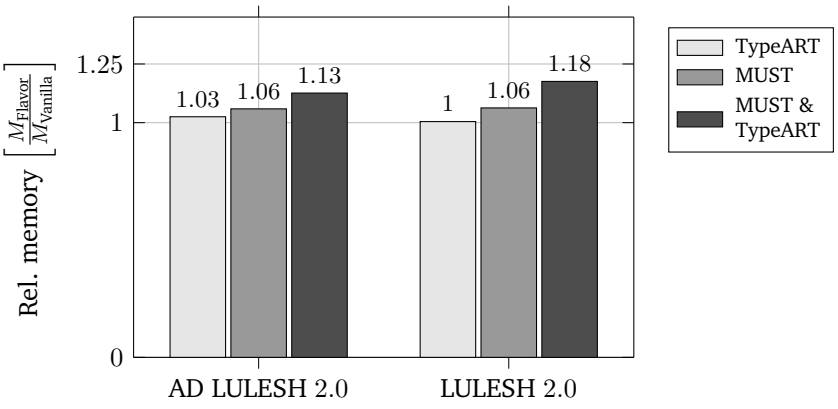


Figure 5.13: The relative median memory overhead of a single MPI process w.r.t. vanilla. Vanilla AD RSS: 150 MB. Vanilla primal RSS: 122 MB.

5.6 Discussion

Correctness for MPI applications benefits developers independent of any AD-enhancement. Questions about the correctness of the implementations can be partially answered with a tool like MUST, and runtime type related errors are fully revealed in conjunction with TypeART. For the process of AD augmentation in particular, two aspects have to be verified. First, is the MPI communication prior to any type change correct, and, secondly, will the code be correct after applying the AD overloading tool. For the latter, as shown in Chapter 4, type related errors may appear after the AD type change. The same is true for MPI communication, and especially in large code bases, where current developers may not have a full picture of any layout assumption or pointer arithmetic applied to MPI-related memory addresses (see discussion further below regarding defects). Thus, even if the program compiles after adding the AD type and the AD-specific MPI libraries, subtle memory related errors can still occur.

The evaluation of the adjoint LULESH 2.0 code has shown that the application characteristic w.r.t. memory operations significantly changes. Many of the additional instrumented memory operations are not part of the MPI communication and, thus, were missed by the filtering mechanism of TypeART. Improving the filtering mechanism is, hence, worthwhile. In addition, MUST is unaware of the overloaded adjoint MPI routines. Thus, MUST checks the vanilla MPI communication routines invoked internally by MeDiPack during forward and backward evaluation. However, the analysis of the data type layout of the buffer passed to the adjoint MPI call should be sufficient to ensure type correctness of the MPI application: If the type (layout) of the passed buffer is incorrect, the internal MPI communication of MeDiPack is likely also incorrect. On the other hand, if the passed buffer is correct, assuming MeDiPack is sufficiently tested, the internal MPI calls are likely also correct. Overall, this reduces the amount of MPI checks to that of the primal application.

Looking at the overall HPC landscape, a study [81] of about 100 MPI applications was conducted. It appears that (1) most codes are limited to MPI 2 and lower (80%), and (2) point-to-point and collectives are the majority of used features. However, relevant to TypeART, it was also shown that derived datatypes are the third most used feature. And especially the larger MPI applications use more (unique) MPI features.

Hence, TypeART will only gain more importance in the future. An AD augmentation requires developers to wrap all communication routines and, subsequently, (re-) verify the layout assumption of the transferred data — previous, legal memory layout assumptions may no longer be valid. In addition, the majority of applications appear to still use an

older standard [81]. Here, TypeART will help adoption of more complex MPI features by ensuring the type correctness thereof.

Type contracts The recent addition of type asserts to TypeART are of interest whenever a developer wants to express a certain contract that needs to be met for a code region concerning overall type correctness. A violation of a contract can indicate erroneous program execution and, using the assert, lead to meaningful diagnostics and less error prone code as a result. In the future, this feature will be extended to a more expressive syntax and extended type checking capabilities. In particular, (1) variable assertion levels to control the granularity of checks for, e.g., debug builds. (2) More expressive syntax, similar to Clang attributes.[4] Together with an extended pruning of allocations that need not be tracked for these asserts, the performance overhead should be reasonable. Depending on how the *contracts* [126] are implemented in a future C++ standard, the type asserts are a useful addition to ensure predictable program behaviour.

Coverage The tested codes exhibited no type errors for the specific runtime configurations. For one code, 104.milc, a benign mismatch was found, a struct with two floats (e.g., `struct{float a; float b;}`) is transferred as a pointer of two `MPI_FLOAT`. However, while this is not immediately problematic, such assumptions of contiguous memory may be violated after an AD type change. Hence, at the very least, these locations require a manual review.

A defect in amg2013 reported by static analysis in [28] was not detected by TypeART. Due to the nature of dynamic sanitizers, the tested configurations never hit the specific code path, hence, missing the type error. In practice, the different control flow dependent execution paths may not detect a bug in a code region that is only triggered for certain simulation configurations. A combination of static and dynamic analysis for such type defects seems worthwhile to explore for these situations.

Performance The usage of a btree-based map implementation already gave an inherent runtime improvement compared to the red-black tree-based map of the GNU C++ standard library. The cache-efficiency of the fat btree nodes, which contain more than one value, is the main reason for this. With amg2013, the red-black tree map takes approximately 20s longer than the more cache-friendly btree map for the same configuration.

[4] https://clang.llvm.org/docs/AttributeReference.html

Further improvements to the runtime performance targets the analysis pass and the runtime, respectively. The most promising optimizations pertain to (1) improving the static filtering mechanism, and (2) caching of the runtime pointer values for fast lookups.

Filtering of allocations, so far, is limited to a TU as the data flow can not be followed further for external functions. Generating a whole program (static) call-graph as a preprocessing of the TypeART analysis is a possible improvement. The call graph can be used to identify all functions which have MPI-related API usage, and thus filter out stack allocations more aggressively. Alternatively, LLVM uses the intermodular link time optimization (LTO), which runs at link time in conjunction with the linker.[5] LTO identifies and resolves external symbols, e.g., to eliminate any unused functions. An implementation of an *LTOModule*[6] may be able to resolve usage of MPI across module boundaries. Filtering allocations will reduce both memory and runtime overheads.

A runtime type information caching scheme seems to be a worthwhile optimization for reducing the runtime overhead at the cost of additional memory. As shown with LULESH 2.0, a total of $40{,}000$ MPI type checks were done but on only 16 unique memory addresses. Hence, a fast hash map, as already proposed by [73], for $\mathcal{O}(1)$ lookups, and a fallback to the slower btree map seem to be worthwhile improvements.

5.6.1 Limitations

A limitation of the runtime approach of MUST was shown with the undetected type defect. Here, the focus is on the limiting properties of the TypeART tool only.

The inherent limitation of any instrumentation tool, and indeed of TypeART too, is the requirement of the whole program to be compiled with the aforementioned instrumentation pass. Otherwise, relevant memory allocations w.r.t. MPI will be missed, which also applies to the use of memory allocated in black box libraries.

A limitation of the LLVM type system are unsigned integer types which are represented as signed values in the IR. Hence, type errors related to the signedness of any integer values can not be resolved with TypeART. There is no inherent way with the LLVM framework to resolve this. A remedy is a preprocessing step with the Clang frontend which adds a specific tag to every unsigned variable in the source code. The tag is then transferred to the IR and parsed by our analysis pass for proper type deduction.

[5] https://llvm.org/docs/LinkTimeOptimization.html
[6] https://llvm.org/doxygen/structllvm_1_1LTOModule.html

Finally, support for multi-threaded codes has not yet been implemented, the focus was on correctness checking of MPI-only codes. However, hybrid parallelization using, e.g., MPI and OpenMP/Pthreads is a popular choice in HPC applications [78; 81]. Supporting this programming model requires changes to the runtime implementation. Stack variables need a thread specific handling, whereas global variables and heap memory allocations can be shared between threads and therefore the current global memory allocation tracking scheme can be kept.

5.6.2 Related work

Related work to the TypeART implementation can be divided into two main categories, (1) MPI type correctness checking, and (2) runtime type tracing tools, which are mainly used to detect, e.g., type confusion detection of erroneous type casts of a value. The latter is mostly done in the context of application security and hardening of applications against, e.g., malicious code execution.

MPI correctness checking Many MPI correctness tools similar to MUST have been described in the past with runtime checking [79; 135] or symbolic execution [27] for, e.g., deadlock detection. The focus, however, is on finding type related errors in MPI applications as this is handled by TypeART.

MPI-Checker [28] uses the Clang/LLVM framework and combines traversing the codes' AST with additional symbolic execution to check for several MPI defects. This approach can not handle the complexity of any potential pointer aliasing, where the initial type of the allocation can not be determined statically, or where symbolic execution is too computationally costly. In contrast, TypeART is resistant to pointer aliasing.

MPI-CHECK [40] finds defects in Fortran codes with compile-time and runtime analyses. To detect type related errors, it instruments target codes to track datatypes and buffer sizes of arguments to MPI calls. Type mismatches between the static MPI datatype of the MPI routine and the Fortran buffer type can be checked for. Derived data types are checked by storing all names passed to `MPI_Type_commit`. An error is reported if the MPI datatype of an MPI call does not match a built-in or one of the stored names of the derived data type. Fortran 90 intrinsics can be used to check for buffer sizes, and are used to check if a receive buffer is large enough. Unfortunately, for C/C++ programs the lack of such intrinsics necessitates explicit tracking of allocations.

In [140], the authors use static analysis and a LLVM-based symbolic code execution framework [20] to find MPI defects. In particular, for buffer type matching, the authors track for each creation of MPI derived types the primitive type components by a backward slicing of the program state. A lookup map with the primitive type components of each derived type is created this way, and used to compare the C buffer with the statically declared MPI data type. This approach fails, if the alias analysis of the backward slicing does not correctly detect the creation of the type or the type of the typeless C buffer is not correctly detected.

Runtime type tracing Runtime type tracing tools based on Clang/LLVM track various aspects of type information for runtime checks. In particular, these tools typically have a similar approach as described in this chapter, i.e., instrumentation of a target code to track the necessary metadata of memory allocations.

Type confusion error sanitizers are a popular application of these schemes. *Caver* [86] is such a tool, it detects type confusions errors in C++ codes for each executed typecast applied to a value in the program. Caver collects type (relationship) information and instruments typecasts and memory allocation operations during the compilation. A runtime system tracks heap and stack memory allocation with a disjoint metadata scheme, partially based on red-black trees, to check for runtime type mismatches. Other tools [29; 48; 73] target the same class of errors with similar approaches. None of them, however, offer checks for MPI (derived) datatype errors.

6 ProAD Framework: AD-aware profiling of codes

The initial problems of a type change to the AD overloading type have been described in Chapter 4. Aspects of correctness of codes using MPI were subsequently discussed in Chapter 5. In this chapter, a work in progress framework to help AD experts with performance analysis called *ProAD* is presented.

The overall goal of the ProAD framework is to provide the basis for advanced AD domain-specific analysis of C++ codes based on the LLVM compiler framework. It combines static code analysis to build a computational graph for each function in a target code with additional structural information, i.e., loop blocks.

The computational graph can be queried for metrics, e.g., how many operations divided among how many statements are applied to input and output values of some function. Further heuristics and analyses can use this information in order to make a decision about, e.g., preaccumulating the derivatives. Thus, the framework aims to build the basis to give developers insights and metrics of a target code's functions and, on a lower hierarchy, loops. This facilitates the application of advanced AD techniques to exploit the analysis results, and eventually allow the automation of the required source changes.

In Section 2.3, the hierarchies in codes for optimal exploitation for AD were already discussed. In this chapter, these concepts are expanded upon in the context of AD profiling. To find regions for special treatment with AD, domain-specific application insight is required, which no standard profiling tool provides. To that end, performance analysis must factor in, e.g, (1) data flow w.r.t. active variables, (2) code patterns where, e.g., many inputs are mapped to one output, and (3) other structures where exploitable hierarchies are revealed.

The rest of this chapter is structured as follows. In Section 6.1, performance analysis and performance engineering are briefly introduced. In Section 6.2, the underlying concepts

of the profiling framework are presented. Section 6.3 and 6.4 present case studies of the exploitation of structure for AD in different contexts. In Section 6.5, an algorithm to automatically find relevant patterns w.r.t. AD in the computational graph is presented. Subsequently, the algorithm is evaluated on the compute parts of LULESH 2.0 and selected kernels of the Minpack [9; 25] test suite. Section 6.6 briefly describes the implementation details of the framework. Finally, Section 6.7 discusses current limitations and related work.

6.1 Performance analysis

Performance analysis of a target application usually focuses on identifying so-called hot spots, i.e., regions of code where most of the computational runtime resources are spent on. For AD, the runtime overhead and, especially for the RM, the additional memory overhead generated by the tape for the backward section is of interest. Subsequently, the collected performance-related metrics are used by a performance engineering expert to reduce the overhead of the identified hot spots.

Methodology and requirements Several performance analysis methodologies exist, e.g., [44] to make the overall analysis workflow efficient and goal-driven. They differ in their approach of giving a starting point and guidance of eventually finding the root cause of a suspected performance defect. Nevertheless, the performance engineer commonly ends up with a hypothesis of the cause of the performance defect. Subsequently, tools (e.g., [1; 77; 109]) can be used to verify the hypothesis, and, finally, remedy any identified problem.

Requirements for a proper performance analysis (tools and workflow) are as follows. Measuring and applying optimizations and tuning to the target code happen early in a project development lifecycle [122].[1] In addition, for tuning, a search direction (based on empirical evidence) to find the root cause of performance defects or determine hot spots is needed [44]. Finally, the tuning of any identified candidate code region must be done for the common use case thereof [122].

On an application level, traditional performance analysis with a profiling tool is either done with (1) code instrumentation [26], or (2) sample-based profiling (short *sampling*), through, e.g., periodic interrupts [95].

[1] This is unlike the case of the common anti-pattern of *premature* optimization [64].

Instrumentation Code instrumentation transforms the target code by introducing callback functions (called *hooks*) in order to generate information about the executed code. These functions are inserted at, e.g., the function entry and exit points, and are used by profiling libraries to collect performance data for that code region.

Sampling Sampling typically uses periodic interrupt events during the code execution and requires no modification to the code. In contrast to instrumentation, it generates a statistical overview, e.g., where most time is spent is indicated by the number of samples for that region. Sampling does not hinder optimizations, as the code is left untouched and can, thus, induce significantly less overhead compared to straightforward instrumentation. Function names to correlate the samples with the target code are extracted from, e.g., debug symbols of the binary. On the other hand, for a "bursty" application with long periods of little or no workloads, the sampling rate needs to be high in order to accurately assess the target which can increase runtime.

Off-CPU Applying the above techniques is commonly done to generate profiles of the user application functions. In contrast, the so called "off-CPU" analysis [43] identifies threads in user applications that are idle, i.e., waiting on a syscall for, say, an IO operation. Hence, the analysis focuses on blocked threads, rather than the application performance of them when running. Like traditional application profiling, the off-CPU analysis can be implemented by either (1) instrumentation, where the syscall entry and exit point in the application are traced, or, (2) sampling, where application threads are, e.g., periodically checked for being off-CPU.[2]

6.1.1 Performance engineering workflow

The performance engineering workflow can be divided into an iterative tuning cycle. In [70] a five stage tuning cycle is described, comparable to the drill-down analysis [44], where the analysis moves from a high-level view to deeper details:

1. **Instrumentation** The target code is primed with instrumentation hooks.
2. **Measurement** The target code is run and measurement data is collected.
3. **Analysis** The collected data is analysed postmortem.

[2] http://www.brendangregg.com/offcpuanalysis.html

4. Understanding The source of any performance defect is reasoned about.

5. Tuning Any performance defect is removed. A validation of the changes is done with a new measurement.

For sampling the first stage can be omitted.

Instrumentation overhead The instrumentation hooks for most of the major C++ compilers can be added by passing a specific flag.[3] Using the compiler, instrumentation hooks are added indiscriminately to every function's enter and exit points. This can induce significant overheads as, for instance, data accessors are quite common in object oriented codes.[4] They do little work and are typically inlined. However, this optimization is likely to be inhibited by the instrumentation. The code, thus, runs much slower. This leads to the point where the application profile is useless as the runtime perturbation can lead to a magnitude of additional runtime [87]. Hence, to keep the overhead manageable, a selective instrumentation strategy is needed. Such a strategy, however, can be time-consuming to realize. Each application requires different strategies to keep overhead manageable while still capturing the needed performance-related details.

6.1.2 Performance engineering for algorithmic differentiation

Performance improvements of AD codes can either be achieved by (1) improvements to the primal code, or by (2) finding AD induced code hot spots and treating them.

Any improvement of the primal can lead to a runtime or memory overhead improvement of the AD enhanced code. Reducing the computational steps to reach a solution of an iterative solver, say, reduces the computational steps on the tape with the black-box AD approach (no special treatments of the solver). In this chapter, however, the latter approach to identify AD hot spots is of concern only. In addition, the aforementioned *off*-CPU analysis is not a focus of this chapter.

[3] e.g., `-finstrument-functions` [41], for the Clang compiler also `-fxray-instrument` [12; 139]
[4] Functions that for instance only set or return a value of an object variable.

Domain-specific properties To gain an understanding of hot spots and potential remedies, AD domain-specific properties of code regions are required. Useful properties for AD are gained through analysis of code regions for, among others, (1) input and output data flow, (2) the computational complexity, as, e.g., many complex statements significantly increase the tape size, (3) code patterns, as, e.g., loop-based sum accumulation can be treated efficiently [52]. If this information was available to an AD expert, they can be used for special AD handling using the concepts presented in Section 2.3. The AD domain-specific performance analysis, thus, does not diverge from the principle tuning workflow of measurement, understanding and improving. Rather, it is a supplemental analysis, as traditional profiling is able to uncover a different class of performance defects. In particular, a performance bug of the original ADOL-C utilization in ISSM was detected by using statistical sampling [107]. The sampling-based profile showed that the function consuming most CPU cycles came from the search for contiguous memory in ADOL-C. For more details on the impact see Section A.2.2.

Existing AD profiling With existing AD overloading-based hot spot analysis, the program is executed with a special diagnostic mode or diagnostic AD overloading type. A combination with function instrumentation allows correlating the data flow on a function level. For instance, the entry and exit hooks can be accurately correlated with the size of the tape before and after the function invocation, respectively. Hence, tape memory size, input and output data flow and other properties can be gathered from the sectioned tape on a function call level specific to that AD tool [90; 117; 130]. Sampling, in contrast, does not allow for this precise and deterministic AD data collection.

dco The AD tool dco uses an "instrumentation mode" with compiler function instrumentation. The instrumented functions act as markers for the tape, sectioning it in order to accurately assess the AD-related impact of any instrumented function [90; 130].

In particular, the special mode stores all function calls ordered according to the call sequence to produce the dynamic call graph. For each function exit event, at runtime, the respective function name and AD-related counters are stored [130], including (1) input/output data counts, (2) number of tape entries (i.e., the tape size), and (3) the number of sub-functions and the cumulative tape size of these.

CoDiPack The AD tool CoDiPack, like dco, uses a special diagnostic AD overloading type to collect AD-related runtime properties for later analysis of a target

application. The authors claim to have improved upon the concepts of dco by making the diagnostic type more time- and memory-efficient, sufficient for "arbitrarily large cases" [117].

The diagnostic overloading type uses the overloaded operators to track at runtime the amount of (1) actives, (2) passives, (3) constants and (4) statements. The counting is done on a function level. With compiler instrumentation, for each entry a statistics tracking object is generated and the overloaded operators increment the aforementioned four counters. Each such event enumerates the function object with a global counting unique id. Then, when the function returns, for each exit, the tracking object is directly serialized to a file.

Tracking inputs and outputs of a function is done with index-based barriers. That is, with AD overloading, typically each value has a specific index on the tape when it is first instantiated.[5] The authors exploit this by also using this index when a function is called, i.e., all indices of the overloading type used inside a function smaller than the function value are instantiated before the function call and, thus, considered inputs. The same concept is applied to output values of a function. The output count is added cumulatively to the callee of the current object at the respective function exit event.

The serialized data, after the target application is finished, represents the dynamic call graph with augmented function-level statistics for each node of the graph.

6.2 Profiling for algorithmic differentiation

The foundation of the profiling framework and fundamentals of AD related analysis is presented in the following section. The implementation details are discussed in Section 6.6.

The framework is based on the LLVM compiler infrastructure and employs the underlying IR (a static SAC) for the static analysis. The static analysis parses each function, and based on the IR, builds a computational graph forming a Directed Acyclic Graph (DAG) with additional structural information. This extended DAG is the basis for further detection of AD-related optimization opportunities by applying heuristics or complexity models specific to AD.

[5] This is tape-implementation dependent but is employed for some tape configurations in CoDiPack and also the diagnostic type.

6.2.1 The computational graph

First, the computational graph in the form of a DAG is introduced. This is an extensional view of the SAC which holds for the LLVM IR. In addition to the expressions forming the SAC, the graph representation of the framework also includes for each function (1) structural information, i.e., `for`-loops, represented as sub-graphs, and (2) every such (sub) graph has optional meta information of the loop iteration counts and interface width.

The computational graph is based on the data flow of floating-point input and output values of the respective function. It includes all operations applied to these values. Input values are either argument values or class variables used inside the function. Output values are defined as either mutable arguments or class variables that are modified inside the function. Hereafter, the inputs and outputs are called *loads* and *stores*, respectively.

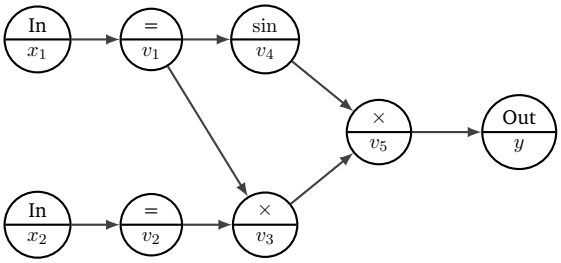
If a function has only loads or stores on non-float values, no computational graph is generated. The function is deemed irrelevant w.r.t. AD computations. Likewise, if a mixed argument set of float and non-float is present, only the data flow of the float values is used to generate the computational graph. Based on the data flow of the float values, some IR instructions may therefore not be in the graph.

DAG and algorithmic differentiation Until now, the implementation of AD was limited to the view of SACs, see Section 2.1. The application of AD can also conceptually be viewed on a computational graph, which can be reconstructed from a tape trace for the RM [90]. Without a formal graph theory introduction: The nodes of the DAG represent the elemental functions and the edges are the flow of the data, i.e., the input and output of the elemental function. A computational graph is, thus, the data dependence mapping from input (independents) to outputs (dependents). In the following, based on the equation (6.1), the DAG, with the two modes of AD applied to it, is given in Figure 6.1.

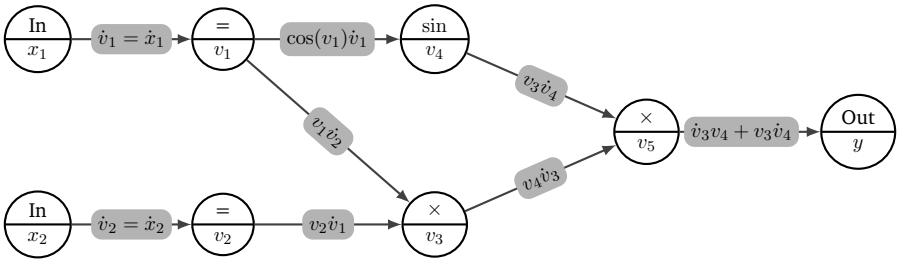
Extension of the DAG with code structure The introduced DAG, so far, represents a compound function consisting of several elemental functions. However, real codes consist of functions with (1) loops or loop-like structures, (2) conditional code paths, and (3) also (potentially deeply nested) calls to other functions during the code execution.

These properties need to be integrated into the computational graph of the framework for accurate AD specific performance modeling.

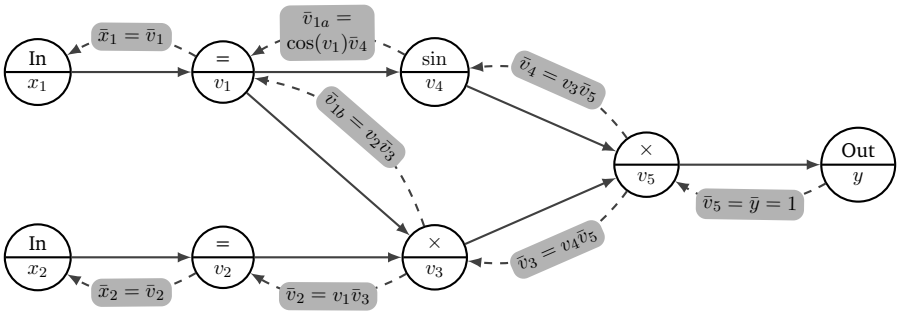
$$
\begin{aligned}
v_{1\ldots2} &= x_{1\ldots2} \\
v_3 &= v_1 v_2 \\
v_4 &= \sin(v_1) \\
v_5 &= v_3 v_4 \\
y &= v_5
\end{aligned}
\tag{6.1}
$$



(a) The DAG of the example function (6.1) (data flow and execution order left to right).



(b) FM: The derivatives are passed along the edges with the forward evaluation.



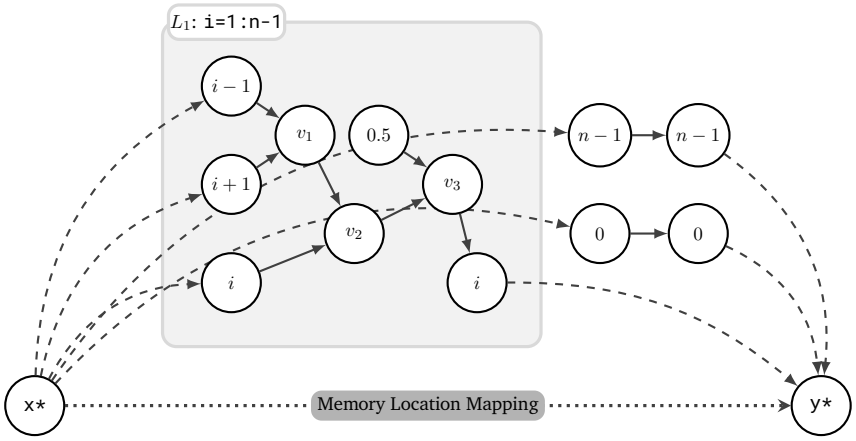
(c) RM: The adjoints are propagated backwards (dashed arrows) after the forward evaluation. The adjoint values $\bar{v}_{1a}$ and $\bar{v}_{1b}$ are added together for the final adjoint $\bar{v}_1$.

Figure 6.1: (a) The DAG of the function $y = x_1 x_2 \sin(x_1)$ and (b), (c) the main modes of AD applied to it. A node represents some operation and has a name. Each node is connected w.r.t. its data flow and passes it along the arrow direction.

Loops Loop-level structural information provides an insight about optimization effectiveness of manually treating a loop. Consider the code in Figure 6.2a and the corresponding computational graph, augmented with the loop structure, in Figure 6.2b.

```
1  void f(const double* x, double* y, int n) {
2    for(int i = 1; i < n-1;++i) {
3      y[i] = .5*(x[i-1] + x[i] + x[i+1]);
4    }
5    y[0] = x[0]; y[n-1] = x[n-1];
6  }
```

(a) A 1D stencil loop.



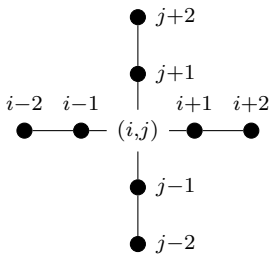
(b) The DAG with additional loop structure. Each dashed arrow is a load or store on x or y, respectively. The solid arrows are the data flow of values. The other nodes are the elemental functions. Statically, the loop bound of $n-1$ and the static DAG of the loop body are known.

Figure 6.2: The DAG is augmented with loop information.

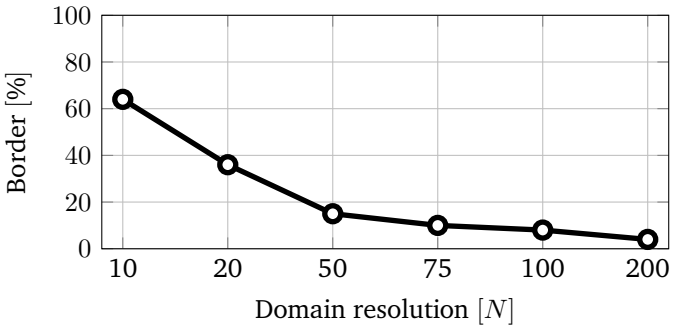
Statically, the loop bounds are unknown and can only be speculated about. However, the loop structure is useful to gather information about (1) the *interface width*, i.e., how many loads and stores are executed inside the loop, and (2) what kind of pattern is present (here a reduction) which can be exploited with AD.

Conditional code paths Conditional branches are commonly found in codes. This also holds for stencil codes: The stencil value at the border of the domain is computed conditionally based on the border conditions of the discretization. At the same time, the border condition specific code path is executed only a few times compared to the full stencil computation inside the domain, see Figure 6.3.

Hence, in accordance to the guideline of “optimizing for the common case” [122], taking



(a) A 9 point stencil.



(b) Percentage how often the border condition is triggered relative to the full stencil on the inner grid region for a domain with $N \times N$ grid points.

Figure 6.3: On a 2D domain, with a sufficient grid resolution, the border conditions will be executed less than 5% of the time.

the branches with the most data flow w.r.t. inputs and outputs of a function is the chosen heuristic.

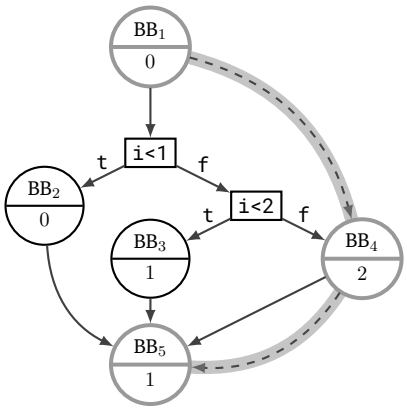
The Control-Flow Graph (CFG) of a function is used to determine the code path with the most data flow measured with the number of loads. A node in the graph represents a block of code. An edge represents the conditional flow during execution. Each node holds the number of data *loads* on the input/output values. Based on a straightforward pathfinding to determine the longest path on this CFG, the nodes with the most loads are used to generate the final DAG of a function. In Figure 6.4, a stencil code excerpt of the left side on a 1D domain is shown with a stencil width of 5 points. As these boundary conditions are rarely executed, the `else` branch is used for the graph.

6.2.2 Heuristics on the graph

The graph concepts presented in Section 6.2.1 are the basis for all further analyses of a target code. The static computational graph contains, for each function, all *loads* and *stores* on the input and output values, connected by operations on those values based on the local data flow. The analysis on the static graph, without any dynamic data, is sufficient for several heuristics to find optimization opportunities. Among them are patterns in the graph that indicate which mode of AD is most efficient for a section of the code.

```
1      // ... loop body ...
2      // ... left border condition ...
3  BB1 double lh1;
4      double lh2;
5      if(i < 1) {
6  BB2   lh1 = const_b;
7        lh2 = 0.5*const_b;
8      } else if(i < 2) {
9  BB3   lh1 = a[i-1];
10       lh2 = 0.5*const_b;
11     } else {
12 BB4   lh1 = a[i-1];
13       lh2 = a[i-2];
14     }
15 BB5 double lh = lh1 + lh2;
16     b[i] = 0.5 * (a[i] + lh);
```

(a) Some loop stencil with border conditions (left domain side). BB_i denotes each basic block, a set of expressions without control flow.

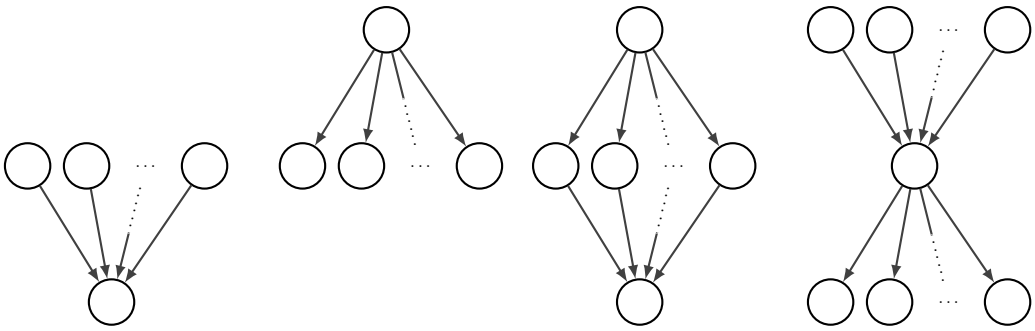


(b) A block has a number of loads, depicted at the bottom of a node. The path of blocks with the maximum number of loads is used: $BB_1 \mapsto BB_4 \mapsto BB_5$

Figure 6.4: Statically, the framework resolves control flow by taking the branches with most overall load and stores to generate the final graph.

Pattern in the graph Finding AD-related optimizations can be based on finding certain graph patterns. In Figure 6.2b, a reduction pattern in the loop is present with three inputs mapped to one output in the loop body. This pattern, for instance, hints at the local application of the RM. Patterns that may be present in codes are (1) *reduction*, as shown, (2) *trumpet*, a mapping from one variable to many, for the FM, and (3) *mixes* of the former two, e.g., a graph which maps from many to one and back to many [14; 18], see Figure 6.5.

Loop based operations with specific properties can be handled efficiently. In [52], "additive reductions" — loops computing global sums — have been identified as targets, i.e., computing the adjoint values (backward section) directly with the original loop execution, thus, lowering overall memory requirements. In addition, [90], Section 3.3.2, shows a sum reduction treated similarly: Knowing the symbolic derivative of a sum reduction in a loop for $y = \sum_i x_i$, the AD expert can manually put the respective adjoint statements on the tape for each iteration of the loop. This minimizes the amount of intermediate statements on the tape due to overriding of the local accumulator in each iteration.



(a) Reduction pattern (b) Trumpet pattern (c) Mixed pattern (d) Mixed pattern

Figure 6.5: Examples for distinctive patterns as indication for a particular suitable AD mode, (a) RM, (b) FM and mixed mode AD, (c) FM then RM, and, finally, (d) RM then FM. Each of the respective patterns have several variations. For instance, a reduction can be a sum reduction on a single accumulator memory location, or, as shown with stencil operations, the same pattern repeated many times (in a loop body) mapping to different memory locations.

If a sum operation is, for instance, present in the computational graph, it can be detected by finding sub-graphs that map several inputs to a constant memory address. The pattern has a similar form as shown in the loop block of Figure 6.2b, except for the constant output variable. On the other hand, a trumpet, as noted by [45], may be less detectable at compile-time. Patterns suggesting the mixed mode, have many variations and detecting all of them requires heuristics. One such heuristic implementation is discussed in the Section 6.5.

Such pattern recognition, especially w.r.t. reductions, has been discussed in context of (general) compiler-related optimization of primal code by others, e.g., [53; 93]. Detected reduction patterns can be optimized by (manually) using special routines in the context of MPI or, locally on a compute node, with compiler vectorization [8] or OpenMP [22] routines. Hence, reductions are of general concern for optimal code generation and, thus, optimal performance.

Preaccumulation In particular, preaccumulation is the precomputation of the Jacobian for a code region during the forward section of the RM and only storing the computed Jacobian on the tape instead of the individual statements/gradients. Preaccumulation is only worthwhile in specific instances. The size of the tape for the individual gradients

must be larger than the size of the tape for the Jacobian with some bookkeeping data. In practice, the numbers of inputs n and outputs m must be small compared to the amount of operations executed (intermediates) for the respective code fragment [45; 117].

Interface Contraction Interface contraction, a category of preaccumulation [14; 18; 45] as introduced, describes embedded code regions of a larger global context with local input count $\hat{n}$ and local output count $\hat{m}$ w.r.t. the global input count n and the global output count m. Then $\hat{n} < n$ and $\hat{m} < m$ indicates an interface contraction. Finding such instances requires a structural analysis of the code [92].

With vector modes of, e.g, FM AD, see Section 2.1.1, sub-functions $\widehat{f} : \mathbb{R}^{\hat{n}} \mapsto \mathbb{R}^{\hat{m}}$ of a target code $\mathbf{y} = f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ that have these properties of a narrowing interface are treated as follows [14]. Instead of passing the seed vector $\dot{\mathbf{x}}$ with length n, a local seed vector $\dot{\mathbf{x}}_{local}$ of length $\hat{n}$ is created to compute the full local Jacobian. The local Jacobian can then be reintegrated into the global derivative computation by a multiplication of the local Jacobian with the global seeding, i.e., exploiting the associativity of the chain rule (2.4) [19; 92].

6.3 Case study: Flow in a driven cavity

This section is an example of a hierarchical approach of AD based on the driven cavity problem of the Minpack test suite [9]. The stencil code solves a flow problem in a driven cavity on a discretized domain with border conditions using a 13 element stencil. The domain has a size of $n_x \times n_y$ elements and also computes the Jacobian $J \in \mathbb{R}^{n \times n}$, $n = n_x \times n_y$ using hand-derived code. In [14], the code was already presented as a case of interface contraction of the stencil loop body in the context of source transformation AD. It is now revisited in the context of a pattern search in the computational graph. The code was ported from Fortran to C++ and CoDiPack was applied to it. The case of the interface contraction is illustrated in the pseudo code of Figure 6.6.

Globally, the data mapping of the whole stencil operation is $\mathbb{R}^n \mapsto \mathbb{R}^n$. However, with knowledge about the underlying code structure, the contraction of the data width can be identified for each element stencil evaluation, and can, thus, be exploited with the RM more efficiently. In [14], globally the code was derived with the vector FM and locally the RM applied to the stencil loop body. The resulting local Jacobian J_{local} is reintegrated by multiplying it with the global derivatives.

```
1  procedure dc:
2    In: X
3    Out: X_new
4    for each n points i of X:
5      N_e(1:13) = neighbours_of(i);
6      x_i^new = f_stencil(X, N_e);
                 R^13 ↦ R
```

Figure 6.6: Driven cavity stencil loop. Globally the function is a mapping $dc : \mathbb{R}^n \mapsto \mathbb{R}^n$ but in the loop body, for each grid point, the stencil evaluation can be interpreted as a local function $f_{stencil} : \mathbb{R}^{13} \mapsto \mathbb{R}$.

Applying the framework to the code, the data flow shows the reduction pattern that leads to the revelation of the contracting interface, see Figure 6.7. The statically generated graph represents one stencil data evaluation ignoring border conditions, see Section 6.2.1.

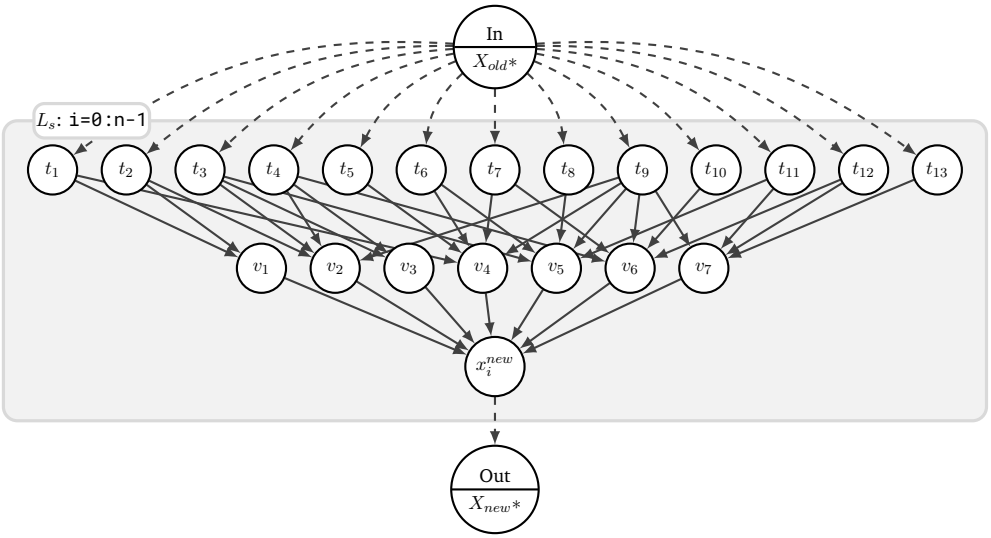


Figure 6.7: Simplified data flow graph (only load and store relationships) of the driven cavity problem: The 13 stencil values (t_i) are loaded and then reduced through some computations to 7 values (v_j) before finally being reduced to calculate the new point `x_new`.

6.3.1 Evaluation

The evaluation of different derivative computations is shown in Figure 6.8. All benchmarks were run on compute nodes of the Lichtenberg high-performance computer of TU Darmstadt, with two Intel Xeon Processor E-2670 at a fixed frequency of 2.6 GHz. The mean of three runs is shown. The standard deviation w.r.t. the mean value was below 2% in all benchmarks. The FM and RM implemented without structural information perform worst for all tested domain sizes. The hand-derived code, on the other hand, is inside the main kernel loop and as such can be optimized by the compiler. Finally, as a showcase of the potential performance improvement by exploiting the structure of the stencil kernel, the optimal RM implementation is shown. It uses the RM for the stencil computation only locally inside the stencil loop body as indicated by ProAD. In addition, the conditional code for the border was extracted into an additional, second loop. This is called *loop splitting*, which is a known compiler optimization technique [33]. As shown in Figure 6.3, the border conditions rarely trigger, compared to the inner region of the discretization. Hence, separating the computation into two loops, one exclusively for the border conditions and one for the inner domain, improves performance. This is reflected with the speedup between a factor of 1.18 and 1.21 of the *RM opt* for the tested domain sizes, compared to the hand-derived code. In addition, compared to the naive RM implementation, speedup factors of 10 to 13.5 are observed.

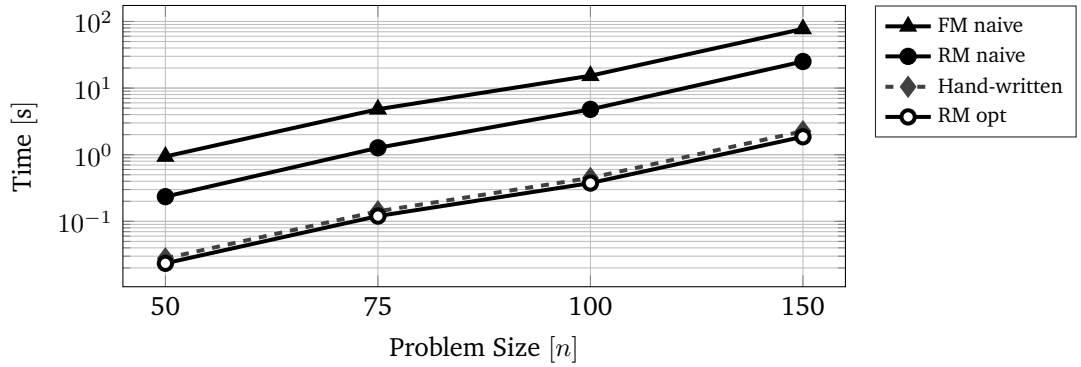


Figure 6.8: Illustration of the performance differences between different approaches to compute the Jacobian of the driven cavity. *Naive*: The differentiation is done as a black-box without structural information. *Hand-written* is manually written derivative statements directly embedded into the stencil computation. *RM opt* is the case of exploiting the structure for RM, including the introduction of an additional loop just for the border regions.

6.4 Case study: Mixed mode algorithmic differentiation

In general, the narrowing of an interface (w.r.t. a code region) can also be followed by a widening and vice versa. Analysis of the data flow and its width is, therefore, a general concern and can lead to a decomposition of the target code into regions where the RM and FM are combined depending on the local data flow. Re-integration of Jacobians computed by different modes is possible with the chain rule, as described with the interface contraction in the previous Section 6.2.2. This, however, comes at the cost of several multiplication operations and, thus, may not be worthwhile for simple codes.

6.4.1 Evaluation: Mixed mode algorithmic differentiation

In [18], it is argued that a graph visualization can aid an AD developer to look for contracting or expanding interfaces. The automatic detection of such narrow interfaces is discussed in Section 6.5.

The example graph In Figure 6.9, the slightly modified computational graph presented in [18] is shown. The input and output width, both 5, suggests the use of the forward mode. However, as evident by visual inspection, the graph has a narrow section at the nodes t_2 and t_5.

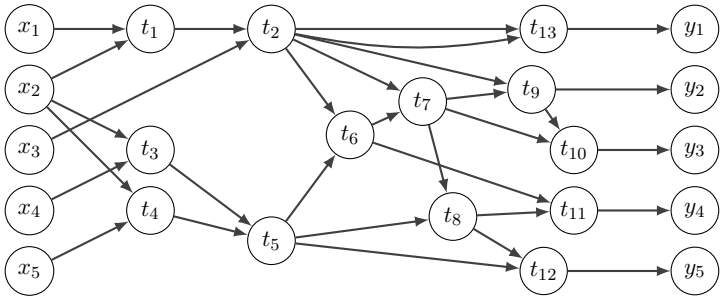


Figure 6.9: Computational graph of a mapping from input $\mathbf{x} \in \mathbb{R}^5$ to output $\mathbf{y} \in \mathbb{R}^5$ with some intermediate nodes t_i, $i \in [1 \dots 13]$.

If the computation is divided at the aforementioned nodes, the first part has input and output dimension $n = 5$ and $\tilde{m} = 2$, respectively. The second part has, thus, input and output dimension $\tilde{n} = \tilde{m} = 2$ and $m = 5$, respectively, see Figure 6.10.

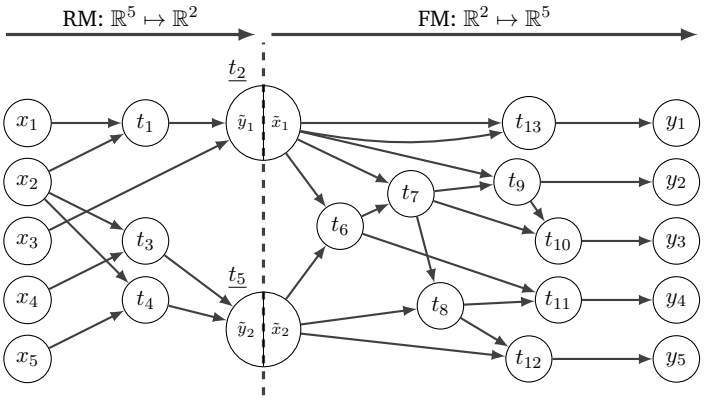


Figure 6.10: The graph is cut at the narrow points. The left side has 5 inputs and two outputs $\tilde{\mathbf{y}} \in \mathbb{R}^2$. The right side has two inputs $\tilde{\mathbf{x}} \equiv \tilde{\mathbf{y}}$ and 5 outputs. In accordance with the theoretical complexity measure of the two AD modes, see Section 2.1, the first half is computed more efficiently with the RM, whereas the second part is computed more efficiently with the FM.

Evaluation with mixed mode A small test code was implemented which structurally represents the graph of Figure 6.9 and the full Jacobian $J \in \mathbb{R}^{5\times5}$ is computed. The nodes t_i with $i \in [1 \dots 8]$ are the result of a multiplication of two values, according to data flow in the graph, e.g., $t_1 = x_1 x_2$. The nodes t_i with $i \in [9 \dots 13]$ mapping to the final output, on the other hand, are a multiplication of the functions $\sin$ and $\cos$. To scale the complexity, each of these operations is implemented as a sum operation, and varying loop trip counts are measured. Figure 6.11 shows a comparison between the (1) FM, (2) RM and (3) Mixed Mode (MM). CoDiPack is applied to the code, and for the MM, the first half of the graph uses the RM type, and the second half is computed with the FM type. The benchmarks were run on compute nodes of the Lichtenberg high-performance computer of TU Darmstadt, with two Intel Xeon Processor E-2670 at a fixed frequency of 2.6 GHz.

The MM is faster than either the FM or RM mode. The standard deviation (SD) for the benchmark of 94 timed runs indicates some jitter. The data was treated by replacing outliers with a linear interpolation of neighbouring, non-outlier values.[6] Outliers are defined as values three standard deviations (SD) from the mean. The outliers of the remaining values are noted. All percentages refer to the SD w.r.t. the mean value. Only SD values above 5% are mentioned. For the MM, the SD is between 12% to 20% for the

[6] https://www.mathworks.com/help/matlab/ref/filloutliers.html

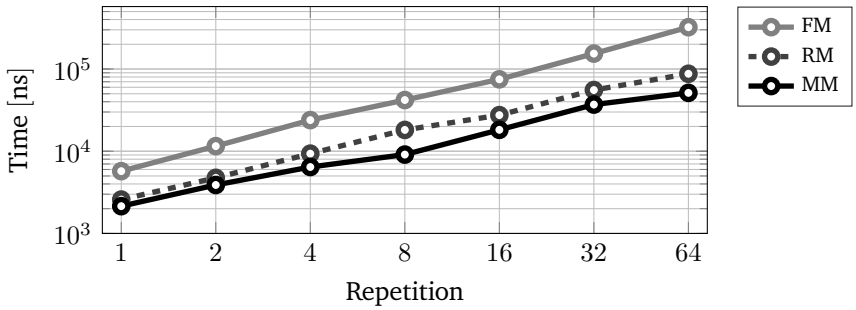


Figure 6.11: Mean time for different AD modes. To scale the computational complexity: *Repetition* is the loop trip count over the respective operation for each node of the computational graph.

first three and the last test size, respectively. Finally, the FM had an outlier for the fourth test size at about 14%. Overall this jittery behaviour can be attributed to the benchmark being small scale.

6.5 Finding cuts in the computational graph

To find these narrow points, shown in the previous two sections, graph visualization and visual inspection can be useful, as shown by [18]. However, for large target codes with many functions, automatic detection is required. Based on the computational graph of the framework, an implementation of an algorithm to detect the narrow points within the graph is presented. Once the narrow points in the graph are found, the detection proposes an appropriate cut as shown in the previous Section 6.4.1.

The algorithm can be understood as a particular variation of the Ford-Fulkerson algorithm (FFA, [35]) and the *Min Cut* problem. The Min Cut is a general term for finding a partition in a graph that is minimal based on a defined criterion [16; 35]. A common application is on flow networks, where the minimal cut are the edges with the smallest total flow whose removal would partition the source and sink of the graph in two parts. In the case of ProAD, the algorithm works on the data flow of the graph and traverses it on a path from source (i.e., *loads*) to sinks (i.e., *stores*). Compared to the classical algorithm of the network flow problem, each edge has a capacity of one and, once a flow (i.e., a path) goes through a node in the computational graph, it is instantly fully saturated (capacity of all outgoing edges is 0), and no further flow through it is possible. Each node in these

paths is uniquely assigned, and the procedure is repeated until no such path can be found any more — the graph has no capacity left for a path to reach a sink, hence, the graph is *congested* at the narrow point. The number of paths indicates the number of nodes in the narrow part of the graph, e.g., for the example graph in Figure 6.10 two paths are found, see Figure 6.13 (b) for two such potential paths computed by the algorithm.

6.5.1 Algorithm description

The overall idea, as described in the previous section, is to interpret the underlying computational graph as a flow network with each edge having a capacity of exactly one. Applying the FFA to it will find the maximum flow in the network. With appropriate bookkeeping, nodes that represent the congestion in the graph (indicating the eventual graph partition) can be identified after successfully applying the (modified) FFA. The algorithm works as follows:

- A supersource node[7] is added to the graph from which all path searches originate.
- The path search works as a depth first search. Whenever a path is found, meta information is added for each node of the path:
  1. The node's state is tagged as visited.
  2. The predecessor and successor of the path it belongs to are stored.
  3. The direction of the edges belonging to the path of the visited nodes are inverted, which creates a backedge for a later backwards traversal (this intuition is taken from the FFA).
- For each node, two visit counts are kept which indicate how often a node has been visited during a forward traversal, or with a backward traversal (using the backedge). These properties are later used to determine the eventual cut.
- Whenever a node is visited, first a check is done if it is already part of an existing path by examining the meta information. Handling path clashes is described in Section 6.5.2.

[7] A node in a graph from which all other nodes in a directed search are reachable.

- The eventual cut in the graph is determined by a failed depth-first search, which is not able to find a new path to a sink. The graph is, hence, congested. Determining the location of the cut is based on the aforementioned visit counts and further detailed in Section 6.5.3.

For the source code of the algorithm, see Section A.4.

6.5.2 Handling path clashes

The path search is random and, as such, one path may visit nodes in a way that blocks another path search to reach a sink without crossing a pre-existing one, see Figure 6.12.

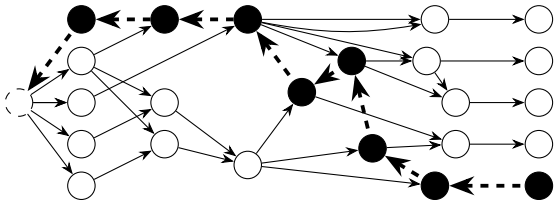
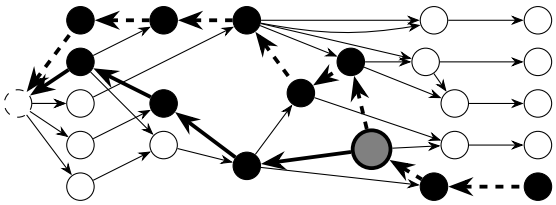


Figure 6.12: Example path from the virtual source to a sink (black nodes). The directed edges are now inverted on the path. These backedges are part of the FFA to maximize flow in a network.

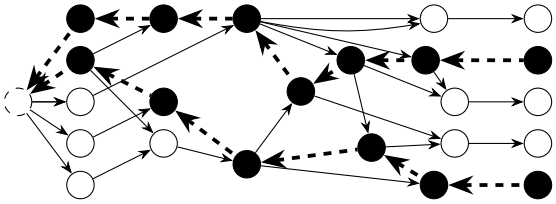
As a remedy, inspired by the FFA, the edges between two nodes that have been visited can be traversed in reverse order, see Figure 6.13. Here, the second path must cross the first path to reach a sink. At the intersection node, the second path is augmented with all nodes of the first path starting at the intersection node to the sink. The search then continues until a new path is found to a different sink: The original path predecessor of the intersecting node is used to continue the search, either forward to a new sink, or along the inverted edges of the now incomplete first path.

6.5.3 Determining the cut

Once all possible paths are determined, i.e., two in the example graph, subsequent traversals can not reach a sink as the graph is congested at the narrow point. Any new path would need to cross the nodes part of an existing path, which are therefore already fully congested. Thus, no new path through the narrow point in the graph exists. All



(a) A second traversal reaches an existing path and cannot cross.



(b) The new path is augmented with the nodes at the crossing to the sink of the original path. The search continues on the previous node of the original path.

Figure 6.13: The second path must cross the existing path to reach a sink (gray node). From this node, the traversal in reverse direction is possible.

nodes from the source to these nodes will be visited by such searches, partly due to the backward edges. All nodes behind the narrow point can not be visited by this traversal, see Figure 6.14, hence, revealing the cut based on the nodes meta information.

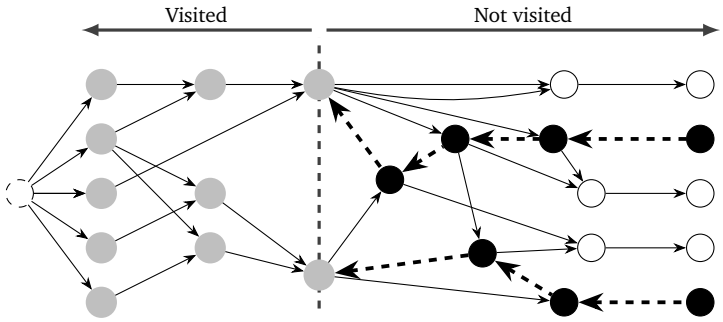


Figure 6.14: A final search attempt originating from the super source can not reach any sink and instead finds all nodes left (gray) of the eventual cut in the graph.

The number of searches is compared to how often a node has been visited either in forward or backward direction (visited using the backedges) during the search. The nodes at the eventual cut have been visited 3 times in the particular example, as 3 searches have been

invoked (two successful path searches, and the final failed one). However, the successors of these nodes have not been visited as much, hence, the nodes are part of the cut.

6.5.4 Requirements for analysis of application codes

In the previous two case studies, two patterns in the computational graph were presented, see Section 6.3 and Section 6.4, respectively.

In particular, the driven cavity problem is a stencil code which, inside the loop body, executes a reduction operation of 13 values mapped to one output value. The cut algorithm finds such reductions, here the narrow point is found at the sink node. However, the driven cavity problem code also reveals several requirements for the detection of patterns.

The driven cavity code is implemented as a main loop nest calculating the updated domain field as well as the Jacobian using hand-written derivative statements. Subsequently, the domain field and the Jacobian is scaled by a constant value in additional loops, see Figure 6.15.

```
1   procedure dc:
2     IN: nx, ny, n=nx*ny, x_old
3     OUT: x_new, Jacobian
4     for (int i = 0; i < ny; i++) {
5       for (int j = 0; j < nx; j++) {
6         // calculcate x_new : ℝ^13 ↦ ℝ
7         // calculate Jacobian
8       }
9     }
10    // Optional:
11    // ... for-Loops scaling the x_new and Jacobian fields ...
12  }
```

Figure 6.15: Pseudo code for the overall structure of the driven cavity code.

Requirements on automatic pattern detection From the above description, several requirements are derived for the automatic analysis:

Per loop Applying the analysis per loop can reveal, e.g., a loop specific reduction pattern as is the case for stencil codes.

Per output In the case of the driven cavity, analysis of the main loop nest exclusively per output variable reveals the stencil update as a reduction operation.

Both of these modes combined are possible with the framework which can generate a DAG on specific outputs and apply subsequent analysis only on specified loop blocks.

6.5.5 Evaluation of the cut algorithm

The cut algorithm is applied to several compute functions of the Minpack test suite and the LULESH 2.0 code base, see Table 6.1.

Overall, the statically detected number of inputs is often higher than the number of outputs. Hence, the reduction-like pattern (an indication for the RM) with a cut as wide as the number of outputs dominates.

Project	Function	In	Out	Cut	AD mode
Minpack	`qform`	6	2	2	RM
	`rwupdt`	12	6	5	MM
	`dfdcfj`*	13	1	1	RM
LULESH 2.0	`AreaFace`	12	1	1	RM
	`CalcElemFBHourglassForce`	14	3	3	RM
	`CalcElemShapeFunctionDerivatives`	24	25	9	MM
	`CalcElemVelocityGradient`	73	6	6	RM
	`CalcElemVolume`	24	1	1	RM
	`CalcElemVolume-pt`†	24	1	1	RM
	`CalcMonotonicQGradientsForElems`	50	6	6	RM
	`CollectDomainNodesToElemNodes`	24	24	24	FM
	`SumElemFaceNormal`	24	12	12	RM
	`SumElemStressesToNodeForces`	6	3	3	RM
	`VoluDer`	21	6	6	RM

*Driven cavity: Algorithm set to ignore the Jacobian computation, see Figure 6.15.
†Consists of a single call to `CalcElemVolume` and returns the (scalar) result.

Table 6.1: Cut algorithm applied to kernels of the Minpack test suite and LULESH 2.0. *In* and *Out* refer to the number of statically detected values going in and out of the function, respectively. *Cut* is the width of the cut in the DAG. The *AD mode* was determined based on the aforementioned numbers. If a cut is, e.g., less than both the input and output the MM is assumed.

Detailed discussion Most analysed functions have a static reduction pattern, which can be handled efficiently with the RM. Potential for the MM was detected twice. However, in the case of the Minpack code `rwupdt`, the cut is marginally smaller than the output. Thus, it is unlikely that the MM gives any speedup compared to the pure RM.

In the following, a subset of the results is discussed in more detail. For each function, a shortened signature is given, including the return value and the parameters.

qform. `void (int m, int n, real *q, int ldq, real *wa)`: The Minpack kernel computes the QR decomposition of a matrix $\mathtt{q} \in \mathbb{R}^{m \times n}$. The code has two outputs, the matrix `q` and a work array `wa` for holding temporary results. In the code, several loops are used to initialize the upper triangle of `q` to constant values. In the ProAD framework, these loops are not part of the computational graph. Only subsequent loops calculating the factored form, which include the operations on the work array `wa`, are identified.

rwupdt. `void (int n, real *r, int ldr, const real *w, real *b, real *alpha, real *cos, real *sin)`: The code works on an upper triangular matrix `r` to which a row `w` is added, including several conditions that hold after the operation, see [25]. The narrow point of the computational graph is 5 as there is a (single) reuse of a scalar variable computed with `r` in a loop which is used for the output arrays `cos` and `sin`, see Listing 6.1.

```
1  void rwupdt(int n, real *r, int ldr,
2              const real *w, real *b, real *alpha,
3              real *cos, real *sin) {
4    ...
5    for (j = 1; j <= n; ++j) {
6      ...
7      if (rowj != 0.) {
8        if (fabs(r[j + j * r_dim1]) < fabs(rowj)) {
9          cotan  = r[j + j * r_dim1] / rowj;
10         sin[j] = p5 / sqrt(p25 + p25 * (cotan * cotan));
11         cos[j] = sin[j] * cotan;
12       }
13       ...
14     }
15   }
16 }
```

Listing 6.1: The `cotan` variable is reused, reducing the cut by one compared to the number of output variables.

CalcElemShapeFunctionDerivatives. `void (Real_t const x[], Real_t const y[], Real_t const z[], Real_t b[][8], Real_t* const volume)`: The function computes some derivatives and stores them in the output matrix (Jacobian) `b`. In addition, the Jacobian determinant is returned as the output `volume`.

The algorithm determines an early cut at the first set of computed variables from the inputs. These are reused for calculating the co-factors for the Jacobian `b` and the output `volume`. Hence, a narrow cut exists that is subsequently widening, see Figure 6.16.

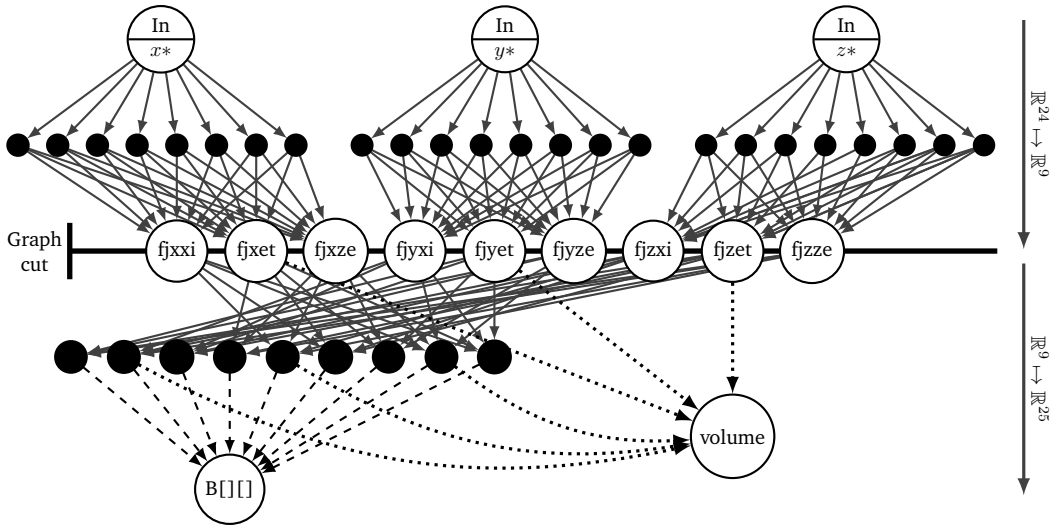


Figure 6.16: Simplified graph for the illustration of the data flow relations: To calculate the partial derivatives, 8 values from each input `x`, `y`, `z` are initially stored in local variables. These are used for computing the additional `fj` variables which are reused to compute the cofactors `cj` for the Jacobian and to compute the `volume` at the end.

CollectDomainNodesToElemNodes. `void (Domain& domain, const Index_t* elemToNode, Real_t elemX[8], Real_t elemY[8], Real_t elemZ[8])`: The function maps several domain nodes to element coordinates based on indices passed to the function. There are in total 24 independent mappings from domain variables to the element arrays, e.g., `elemX[0] = domain.x(elemToNode[0])`. Hence, there is no useful cut determined and the mapping indicates the FM.

6.6 Framework

At the core of the framework lies the computational graph (hereafter *graph*) which was conceptually presented in Section 6.2.1. It is generated during the compilation of the target program using the LLVM compiler framework.

The underlying LLVM IR is canonicalized with some simplification passes provided by the LLVM framework, among them are (1) the promotion of memory to registers (eliminating several load and store instructions which are irrelevant to the graph), and (2) several basic block simplifications. Most of the algorithms of the framework are robust against the form of the code. The normalization is focused on easier user debugging and simplified algorithms.

6.6.1 Analysis pass

The analysis pass processes the canonicalized intermediate code to (1) extract the relevant operations for the computational graph, (2) finds loop-like code regions for additional structural augmentation of the graph, and, finally, (3) assembles the graph using all this collected information.

To that end, the target code is analysed on a function level during compilation. For each function, the input and output parameters are determined, and based on load and store operations, the data flow is followed and all intermediate instructions are collected to assemble the final graph. Loop-like structures are queried for using pre-existing loop analysis in the LLVM framework. The analysis finds loop-like structures in the intermediate code which can be used to determine if any instruction is part of a loop.

Input and output of the graph The computational graph is built for functions with input and output of floating-point values. Otherwise, if the function does not mutate inputs and generates no output, the function is assumed to be irrelevant to AD.

To that end, the analysis is based on the *def-use chain*[8] which is provided by the LLVM framework to follow the data flow of the *inputs* of a target function's IR. All load and store instructions on inputs and outputs of a function are collected. For each detected store, the algorithm assigns a list of the subset of the previously identified load instructions, see Figure 6.17.

[8] https://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains

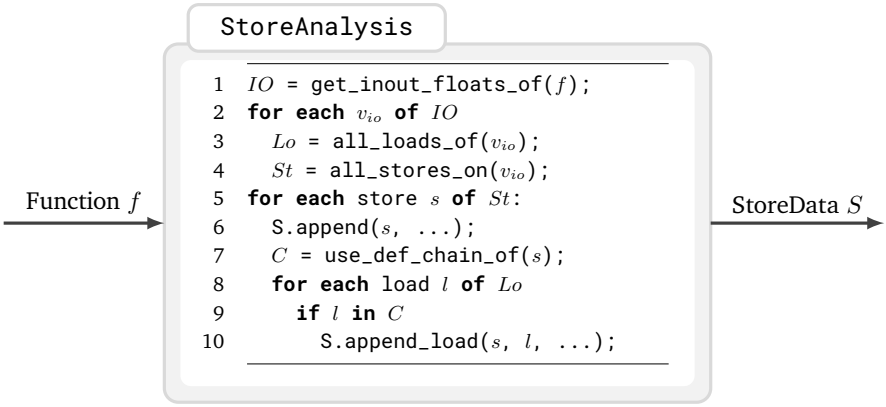


Figure 6.17: The input and output of a function are determined and, based on these variables, the load-store dependency chain is built. IO contains, for instance, arguments to the function f. For object-oriented languages, class variables need to be added to the set. Not shown: Appending to S also resolves dependency of the stores and loads to the function arguments etc.

The store instructions and all connected loads for each function argument are now known. The output, however, is independent of the control flow and shows only the static dependencies between load and store statements without any structural information. Hence, to filter out parts of the instructions that are never executed together, i.e., conditional branches, further analysis is applied, see the paragraph below.

Filtering loads based on the control flow A CFG based on the basic blocks of the function's intermediate code is built. For each node in the CFG, the number of dependent loads is assigned to it for each previously identified store. The path through the CFG with the maximal number of dependent loads is then considered for inclusion into the computational graph, see Figure 6.4. This is repeated for each store of the first analysis step, see Figure 6.18.

Handling backedges Loops in LLVM are given as a set of basic blocks which may have backedges [89] between them, i.e., edges from the loop body to the loop condition and vice versa. The aforementioned longest path analysis (i.e., most loads on a path through the CFG) and the corresponding topological sorting require a DAG to work correctly. Hence, these backedges are handled as shown in Figure 6.19 to gain the acyclic property.

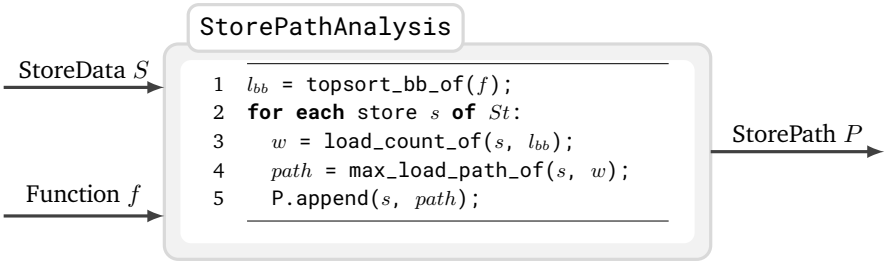


Figure 6.18: The basic blocks of a function are topologically sorted, ignoring backedges of, e.g., loops. Based on each previously identified store, a weight map of the number of load instructions for each basic block is generated. Finally, the path with the maximum number of dependent loads is taken.

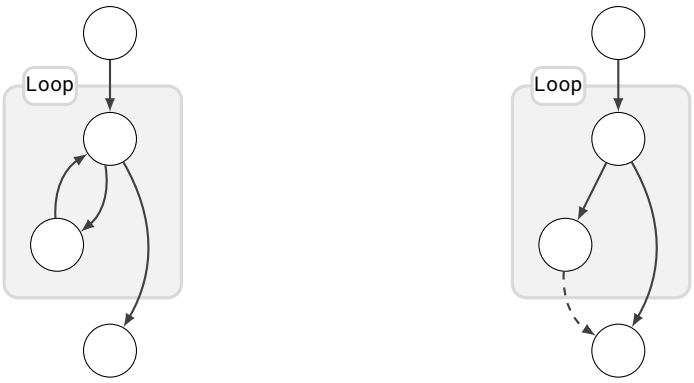
The algorithm applied to a single loop in a function is shown in Figure 6.20. The first node is the entry block, the last node is the exit block. The two nodes in-between represent the loop header and the loop body, respectively.

```
1   procedure add_cfg_edges(BasicBlock* bb_cur):
2     if already visited bb_cur:
3       return;
4     for each successor bb_s of bb_cur:
5       if bb_cur has backedge to bb_s:
6         for each successor bb_ss of bb_s:
7           if bb_cur not reachable over normal edge from bb_ss:
8             add_virtual_edge(bb_cur, bb_ss);
9       else:
10        add_edge(bb_cur, bb_s);
11        visit(bb_s);
```

Figure 6.19: Adding (virtual) edges to the CFG of basic blocks. The algorithm starts at the entry basic block of a function and adds edges to a CFG which contains all basic blocks of a function.

6.6.2 Building the graph

Generating the final computational graph for each function builds upon the analysis results described in Section 6.6.1.



(a) CFG of a single loop in a function. (b) Result of the algorithm of Listing 6.19.

Figure 6.20: Elimination of loop-related backedges for the construction of a directed acyclic CFG. On the right: The virtual edge (dashed) replaces the backedge.

In Figure 6.21, the graph generator is shown, which results in two different graph representations. First, the `Graph` contains all instructions relevant to the input and output of a function. Second, the `BlockGraph` only contains structural information, e.g., it contains loop blocks and the other basic blocks. This way, for each such `block`, block level information are computed, e.g., the local input nodes and the number of statements for such a block.

The underlying data structure for the graphs are the same. Only the node types are different. The data structure uses an adjacency list to manage the edges between nodes. In particular, a node in the `Graph` has a (1) unique ID, (2) an ID to the corresponding block in the `BlockGraph`, and (3) an instruction type indicating the underlying IR code. A node in the `BlockGraph`, on the other hand, stores whether the block is, e.g., a loop and indicates loop nests.

Serialization Serialization is straightforward with the differentiation between code structure and instructions. The trivial graph format (TGF, [115]) is used for the serialization as it is memory efficient and separates data from visualization, unlike, e.g., the dot graph format [31]. The TGF format was already used by others for the same purpose, see [117]. Both of the generated graphs are serialized to the same file, storing the aforementioned data properties for the `Graph` and `BlockGraph`, respectively. Additionally, the framework can also produce graphs in the `dot` format for visual inspection.

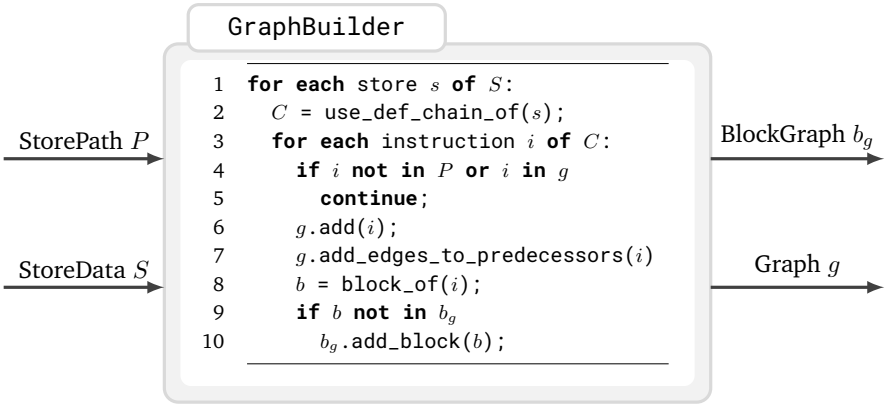


Figure 6.21: The use-def chain is followed in reverse order of the data flow from each previously identified store instruction. Based on the load store relationship in `StoreData`, the filtered path with most number of loads for each store in `StorePath` is used to generate the computational graphs.

Analysis on the graph Once the graph is built, it is the basis of further analysis. The algorithm to find the narrow cut in a graph, see Section 6.5, is one such example. Other analysis relates to finding the number of assignment statement and the amount of active and passive variables w.r.t. AD and constants for each such assignment.

6.7 Discussion

In Section 6.1, requirements on the performance analysis and the tool were presented. Namely, (1) early application of analysis, (2) sufficient tuning directions (e.g., hot spot detection), and, finally, (3) optimization for the common use case. The requirements can be covered by ProAD, especially once further analysis models are explored. Using compiler technology, domain-specific profiling can be applied early, even before the AD augmentation process of the target code is started. Approximations of, e.g., stencil loops with conditional branches fulfil the property of optimization (analysis) for the common code path. The static analysis builds the computational graph relevant to AD for a function. This is already sufficient to do a localized graph analysis to find patterns or complexity metrics that can reveal optimization opportunity as shown with, e.g., the graph in Section 6.4.

6.7.1 Limitations

Static analysis is inherently built upon approximations for undecidable states, such as conditional code evaluation [82]. A case of border conditions for stencil kernels was shown. The framework takes the approach of over-approximation, i.e., the conditional code paths with most load and stores on the interface of a code region are considered, see Figure 6.4. In contrast, this may fail if the actual common code path is dependent on a different metric. For the analysed codes, this was, however, not the case. The AD metrics of active and passive variables w.r.t. AD or statements are, thus, over-approximated in a static analysis context. Symbolic function execution [10] could be explored to go beyond the current employed heuristic. Taking the (symbolically computed) state which is identified as the common code path will more accurately represent the computational graph of a target function.

The static analysis is able to find, e.g., function-local reduction patterns but, as any loop bounds are unknown, the global data mapping pattern is unknown. Likewise, the static computational graph to count the metric of active and passive variables w.r.t. AD has to approximate these numbers as the seeding of values is only known at runtime and would require extensive and accurate analysis of data flow across TUs.

6.7.2 Related work

The related work of AD overloading tools dco and CoDiPack was already discussed in Section 6.1.2. In this section, the approach of the ProAD framework is contrasted against the existing tools.

ProAD approach comparison Both diagnostic modes of the aforementioned tools are not publicly available. Hence, only CoDiPacks diagnostic tape is compared to the LLVM-based ProAD framework as it is described in more detail in [117] compared to dco.

The advantage of the overloading diagnostic approach is that the implementation is straightforward if the target code is made ready for the black-box introduction of the AD type. Hence, only the AD type is changed to the diagnostic mode for collecting metrics. Measuring actives (with an existing seeding routine), passives and constants are only an implementation detail, which, for instance, the underlying diagnostic tape may also provide. The dynamic call graph with function-level AD measurements is

generated by hooks for function entry and exit events added by standard flags for compiler instrumentation.

In comparison, ProAD offers static analysis independent of a working AD implementation for the target code. This allows for a priori knowledge of any optimization opportunity, and exploitation during the implementation and maintenance of the initial type change. The initial type change, as shown in Section 3.1.1, requires changes to many areas of the target code. Knowing about optimization opportunities a priori allows for these code changes to be integrated into the same project milestone.

The overloading approach only generates data for a specific simulation run, as it only collects data for the active code paths. A change of the problem dimension or some other configuration flags may change the application profile drastically. A combination of static analysis and a dynamic data refinement is more robust to such changes as the static approximations are independent of any specific runtime configuration.

In addition, the AD overloading diagnostics indiscriminately collect runtime information. If a function is called many times, with only some slight variations in argument values, say, recollecting the same metrics many times is wasteful. The only interesting metric, after the first call, would then be the total invocation count. With static analysis, the function needs to be analysed once.

7 Conclusion

We conclude with a brief summary of this thesis, highlight our contributions and give an outlook for future work opportunities in the field of AD tooling.

7.1 Summary

The various aspects of software engineering of an AD type change were shown in Chapter 3. An analysis of several AD enhanced codes, all of similar code size and code complexity, in Section 3.3, revealed recurring software changes to integrate AD overloading tools. This included handling of (1) data conversions from the AD type to some other (built-in) type, and (2) integration of external libraries, especially external solvers by symbolic differentiation. In addition, a particular case of exchanging AD overloading tools in the ISSM code base also revealed performance problems of the original AD tool utilization and software bugs in the adjoint MPI library used by the newly introduced AD tool. The similarity of the software changes in these codes and the bugs found motivate the compiler-based tooling approach presented in this work.

Chapter 4 describes problematic code constructs that need to be handled when a user-defined type replaces the built-in floating-point type in a C++ code base. The C++ standard treats these user-defined types differently, hence, compiler errors can occur in target codes due to the AD augmentation. The main issues are the data conversions, either explicit or implicit, that need to be handled by applying appropriate code changes for each such instance. For each problematic construct, we propose solutions and show the static source analysis and source transformation tool *OO-Lint* based on the Clang compiler framework to automatically detect and fix these issues. An application to several scientific C++ codes shows the relevancy, as we find issues in all of them.

MPI, being the de facto standard for distributed computations in HPC codes, has a complex interface which allows usage errors. In particular, when (1) changing target platforms,

(2) adding new communication calls, (3) exchanging new data types, or, (4) in the case of AD, changing the underlying MPI library calls, subtle bugs can be introduced. Our tool *TypeART*, in particular, is used to ensure type correctness of all phases of the MPI communication. It enhances the existing MUST tool, an MPI correctness and deadlock checker. To that end, TypeART tracks type information for all memory allocations (relevant to MPI communication) in a target program by instrumenting these allocation statements using the LLVM compiler framework. The typeless void pointers passed to the MPI library routines, hence, can be checked to detect erroneous memory access patterns. Applying TypeART to several target applications shows its induced overheads to be manageable while, at the same time, having full MPI type check coverage in MUST.

Finally, Chapter 6 describes an AD domain-specific performance profiling framework called *ProAD* based on the LLVM compiler framework. ProAD builds a computational graph based on the data flow of input and output values for each function using static analysis. The graph is further augmented with structural information, i.e., loop-like structures in the code, which allows for a hierarchical view on the graph. The static analysis approach can indicate optimization opportunities without code execution by analysing the computational graph. A graph can have contractions and expansions that can be exploited with a deliberate mix of the two main modes of AD, see Section 6.4.1. The current implementation of the framework can automatically find these narrow points in the graph. Likewise, loop stencils, see Section 6.3, have a specific node pattern in such a graph and, as shown, can be exploited to gain efficiency similar to hand-written derivative code.

7.1.1 Conclusions

Applying AD overloading to complex C++ codes with many advanced abstractions is currently the only feasible way to compute derivatives. AD overloading, applied to the code in a black-box manner, seemingly only requires a change of the underlying type for the computations and localized routines for seeding and extraction of derivative values. However, as shown, the type change to the AD overloading type is often followed by an extensive maintenance of the target code, see Chapter 3. This includes handling of type conversions to accommodate the different semantics of user-defined types in C++, code hybridization, and integration of adjoint MPI libraries for efficient derivative computation.

Compiler tooling can help to remedy some tediousness involved with applying AD efficiently. The approaches presented here help with a subset of the workload to integrate

AD overloading into a target code. In particular, this includes (1) OO-Lint, a tool to fix problematic code constructs w.r.t. user-defined types during the initial type change, (2) TypeART, a tool to track type information for any memory allocations to ensure correctness of MPI communication, and, finally, (3) ProAD, a tool for AD domain specific performance profiling for efficient derivatives. Unlike a full source transformation approach, the focus of the presented tools avoids the inherent complexity of analysing and transforming whole C++ code bases for derivative computations. These specialized tools are, therefore, easier to maintain and extend.

7.2 Outlook

A *support* tooling approach for AD overloading by static analysis in combination with source transformation is, in our view, worth exploring further. First, we discuss potential improvements to the presented tools before showing promising avenues for new tools.

OO-Lint's analysis of problematic code constructs in the aforementioned context of template instantiations needs to be further investigated and subsequently improved.

TypeART for MPI applications benefits from filtering out memory allocations which are not used in the context of any MPI library call. The current data flow analysis is too limited as its context is the current translation unit only. Extending it with a complete (static) call graph makes the filter more effective and, thus, the overhead is minimized further. In addition, tracking memory allocations is useful outside of MPI applications, e.g., the presented type asserts allow a contract based programming style with type-related predicates. These predicates ensure that a developer only passes a set of compatible types to some (generic) API, and, otherwise, a warning or an error is produced if such a type contract is breached. In the context of AD MPI libraries, the MUST tool needs to be extended for the specific semantics of adjoint MPI communication. Hence, MUST only needs to intercept the adjoint MPI calls instead of the corresponding vanilla MPI calls (for the forward and backward section). This way, approximately half of the applied MPI type checks are avoided.

ProAD is still a work in progress. Additional graph-based analysis needs exploration. The implemented graph analysis to find narrow regions in the graph is one such example. However, metrics to evaluate if such a cut is beneficial are yet to be fully examined. In the future, automatic, localized source transformation is worth exploring. For instance, exploiting interface contraction or narrow graph regions with mixed mode AD currently

requires tedious manual code modification by an AD expert. In addition, on a hierarchical level w.r.t. the function call graph of a target code, mixed mode or reduction patterns similar to a stencil can also be found.

In general, there seems to be a wide scope for compiler-based tooling to support the application and optimal usage of AD overloading. For example, to optimize the mechanism behind the expression templates of modern AD tools, statement aggregation through source transformation seems beneficial. Here, subsequent statements are combined to a single long statement and, as a result, the tape related overhead of intermediate statements on the tape is minimized. To avoid making code unmaintainable due to the (long) aggregated statements, these transformations can be done during compilation, just-in-time, without persistent changes to the target code base.

A Appendix

A.1 AD-enhancement of the ULF solver

In this section, the source code changes to the ULF core code are briefly illustrated and discussed, which is an extension of the brief summary given in [61]. The presented aspects are typical for the AD-enhancement of codes.

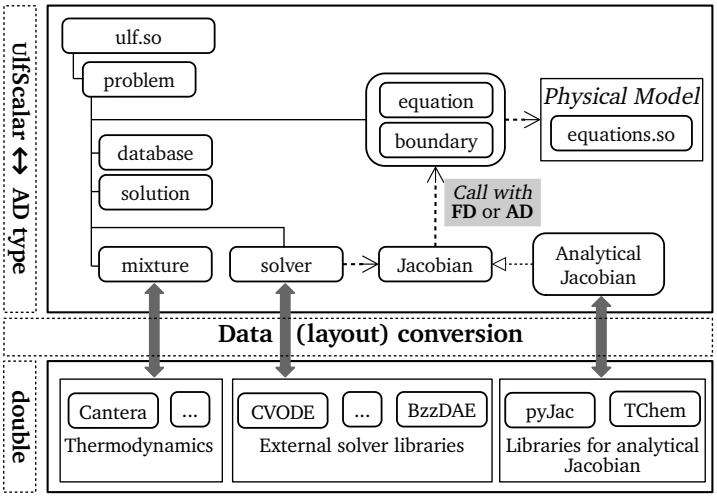


Figure A.1: Simplified structure of the ULF solver with external components.

In Figure A.1, the overall structure of the ULF solver is illustrated. The core library `ulf.so` holds a `problem` class that describes the physical model to be solved, i.e., state equation and helper objects for the solution and physical properties. The `database` manages the required structures and configurations of the problem. The `solution` holds the fields of the current solution of the state equation. Data conversions happen when external

libraries are used. Here, a conversion between the external library specific data layouts and the data fields used in ULF is applied: (1) The mixture class handles thermodynamics (transport properties) of the reactions. (2) The solver wraps external libraries that drive the simulation. These require the current state and Jacobian of the state equation. (3) Code for analytical Jacobians generated by, e.g., pyJac have different internal data layouts.

A.1.1 Preparing the code for algorithmic differentiation

The ULF solver package was written with plain `double` type usage. In order to differentiate the computational parts of ULF, two structural code changes had to be made: (1) The plain double type was replaced with a newly introduced alias `ulfScalar`, and (2) the ULF data structures were made generic, now making use of the defined alias.

The alias, defined in a centralized header, is either set to the built-in double type or a CoDiPack FM or RM AD type at compile time, see Listing A.1.

```
1   using ulfScalar = double | codi::RealForward | codi::RealReverse;
```

Listing A.1: The alias is set at compile time using preprocessor flags for each respective configuration. RealForward and RealReverse are the standard CoDiPack AD types for the FM and RM, respectively.

ULF uses the C library function `malloc` for memory management, which does not invoke the constructor during the allocation of the AD type. Hence, the AD value might not be correctly initialized if it relies on, e.g., tape related book-keeping inside the constructor. The CoDiPack types correctly work with these memory management constructs. Hence, the C library calls were not replaced during the initial changes.

Generic data structures The fundamental numeric data structure for computations in ULF is the `field` class. It stores, e.g., the temperature and concentration of the different chemical species. In accordance to [108], this class was refactored and templatized in order to support generic scalar types, specifying two parameters, (1) the underlying vector representation type which holds the data, and (2) the corresponding, underlying scalar type, see Listing A.2.

This principle is consequently applied to every other data structure in ULF, which have the scalar alias (e.g., `ulfScalar`) as the basic type dependency.

```
1  template <typename Vector, typename Scalar>
2  class fieldTmpl : public fieldDataTmpl<Vector, Scalar> {
3    // mathematical functions, accessors etc.
4  };
5  // alias definition for field in the ULF framework:
6  using ulfVector = ulf::nativeVector<ulfScalar>;
7  using field = fieldTmpl<ulfVector, ulfScalar>;
```

Listing A.2: The class `fieldTmpl` is the base class for fields and provides all operations. They are generically defined in `fieldDataTmpl`. The ULF framework defines `field` as an alias for the templatized data structure based on the `ulfScalar` alias. Similarly, `ulfVector` is an alias for, e.g., ULFs native mathematical vector implementation based on `std::vector` of `ulfScalar` types.

Data conversions With the alias defined as the built-in double, the code was fully operational. However, with the AD type several compiler errors occurred. The origins of these errors are due to the following reasons: (1) Code locations were *missed* during the change to the alias, hence, an AD type was assigned directly to a plain double type. Here, the compiler introduces an implicit type conversion[1] of the form $T \leftarrow \widetilde{T}$. This is only possible if the AD tool supports such conversions with a user-defined conversion function, which the compiler can subsequently make use of in these contexts. However, implicit conversion may lead to an unintentional loss of derivative information and as such need to be handled with care. (2) Broken data conversions between the external libraries and ULF, e.g., the internal Jacobian layout of an ODE solver compared to the ULF data layout. (3) Diagnostics (macros and assert statements for debugging) and routines of the C library for console or file output throughout the code base.

To handle some of these constructs with AD, an explicit value extraction of the primal value is required. To that end, several related templated conversion functions are defined, see Listing A.3.

The caveat of the shown code is the inability to handle CoDiPacks use of expression tree objects. In particular, if the result of any expression is directly passed to the value extraction functions, i.e., `value(x*x)`, no appropriate specialization is found by the compiler. As shown in Section 2.2.2, `x*x` yields a placeholder object, which is only resolved during an assignment. Hence, it does not match any of the specialized functions and the generic

[1] Conversions from type A to type B which are introduced by the compiler without the developer specifying them.

```
1   namespace detail {
2     template <typename T> struct ForSpecialization {
3       static auto value(const T& v) { return v; }
4     };
5     template <typename Tape> struct ForSpecialization<ActiveReal<Tape>> {
6       static auto value(const ActiveReal<Tape>& v) {
7         return v.getValue();
8       }
9     };
10  } /* namespace detail */
11  template <typename T> auto value(const T& v) {
12    return detail::ForSpecialization<T>::value(v);
13  }
14  template <typename To, typename From> To recast(const From& v) {
15    return static_cast<To>(value<From>(v));
16  }
```

Listing A.3: The functions `value` and `recast` (line 11–16) are used for value extraction and type casting, respectively. The former makes use of the struct in the `detail` namespace which is used to specialize for the CoDiPack AD type `ActiveReal`. This is used for the FM and RM. If built-in doubles are used, the value is simply returned. `recast` uses `value` for the value extraction and applies the C++ cast to the template type `To` on the returned value.

fallback is used (line 2–4 of Listing A.3), which is erroneous. These cases were only present a few times in the code, though, and were handled manually. One solution of structuring the value extraction for these cases is given in [117].

The conversion functions are the basis for all other type related changes in the ULF solver, and with appropriate template specialization they work with any type. Notably, they are used for wrapping C library usage for printing (*IO*) and memory related operations (*memcpy*). This design reduces the code maintenance burden of introducing different user-defined types. If, in the future, the scalar alias is set to, e.g., a multi-precision type, only an additional specialization of the aforementioned conversion function in a single header file is required.

External libraries: Solver and analytical Jacobians Two modifications for the solver reintegration were necessary: (1) The data conversion for the external solvers was extended to work with user-defined types, schematically shown in Figure A.2. (2) ULFs custom solver uses data fields directly and lacks a clear separation layer. The code was, thus, augmented with several `recast` operations.

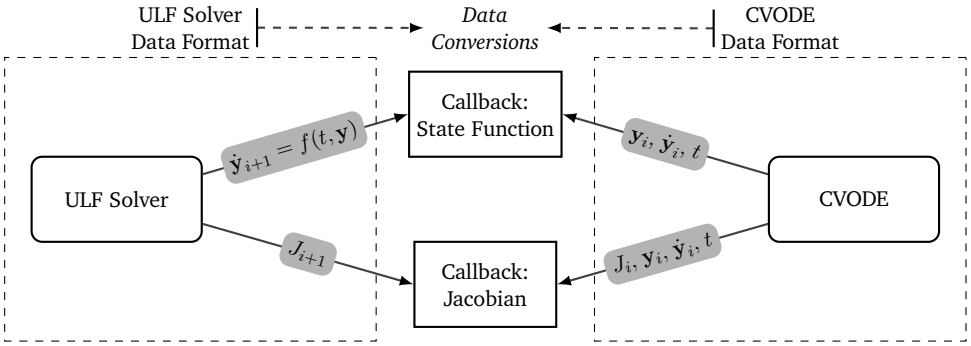


Figure A.2: Solver data flow example, here CVODE [55]. The solver requests function or Jacobian evaluation via callback functions. Pointers for the required internal data structures are passed. (1) During the callback, the data is copied to an ULF internal representation, (2) the respective routine is called, and, finally, (3) the resulting data is transferred back to the CVODE internal format.

Likewise, libraries to calculate the Jacobians analytically have an internal format different to ULFs. Whenever ULF uses these libraries a wrapper implementation is used which (1) converts the current model state to the specific library-dependent format, (2) calls the library to calculate the analytical Jacobian, and, subsequently, (3) transfers the Jacobian to the ULF internal representation.

Copying pointer memory locations To copy arrays, `memcpy` of the C library is sometimes used to, e.g., transfer data from ULF to some external libraries. This is error prone when the source and sink types differ in memory size, as can happen with the user-defined `ulfScalar` and the plain double type of the external libraries. As a remedy, a wrapper is introduced that, depending on the source and sink types, either uses (1) `memcpy` when both types are the same, and, otherwise, (2) a loop-based copy implementation using the aforementioned value extraction functions.

Printing The ULF solver extensively uses the C library for output to the console or a file. For these functions, the arguments specifying the data for printing are expected

to be built-in types and, thus, do not work with the user-defined AD type without modification. The code base, therefore, defines functions which wrap these variadic functions to be compatible with the AD type, see Listing A.4.

```
1  template <typename... Args>
2  void ulf_printf(const char* fmt_string, Args&&... args) {
3    // Call printf with values of expanded "Args"
4    printf(fmt_string, value(std::forward<Args>(args))...);
5  }
```

Listing A.4: The `printf` function is wrapped. Using the C++11 feature of template parameter packs and the respective pack expansion, on each argument the value function (see Listing A.3) is applied. Similar implementations exist for `fprintf` and `sprintf`.

Debugging: Assert and macros The ULF solver defines internal macros and custom assert statements (defined as custom macros) for numeric analysis and debugging purposes during software development. These can be toggled at compile time and are no-ops for a typical simulation run. The (macro-based) code of the assert statements was modified to make use of the `value` extraction functions. The changes, however, are mostly limited to two header files.

A.2 Aspects of the ISSM AD type exchange

This section presents two deficiencies revealed during the ISSM AD type switch [59]:

1. The AMPI library had several bugs which were revealed during the CoDiPack integration test phase, see Section A.2.1.
2. Benchmarks revealed comparatively slow and atypical performance of the original ISSM/ADOL-C implementation. This lead to an effort to identify and, subsequently, to remedy the root cause through sample-based profiling, see Section A.2.2.

A.2.1 AdjointMPI defects

This section summarizes changes applied to the AMPI library by an AD expert during the work of [59].

The AMPI library was the main library for MPI communication when using CoDiPack at the time of the type exchange in ISSM. Since then, it was replaced by a new library called MeDiPack created by the developers of CoDiPack. MeDiPack is used in the current public release of ISSM. Nevertheless, the following described changes were made to AMPI for correct operations in ISSM.

Required code changes ISSM makes use of a subset of MPI-2 communication routines [97], including (1) blocking point-to-point communication (send and receive), (2) and several collective communication routines, e.g., broadcast or gather operations.

In a first integration step, four missing interface definitions in AMPI were identified and added, namely `Allgather`, `Allgatherv`, `Gatherv`, and `Scatterv`.

In addition, ISSM communicates both passive double values and the active AD type using the centralized MPI wrapper. To that end, it defines the MPI datatypes for both in the MPI wrapper. The passive double values use `MPI_DOUBLE`. The active type use the provided definition of the AD MPI library, i.e., `AMPI_DOUBLE`.

However, AdjointMPI assumes that *all* double types in a program, which are used in the adjoint MPI context, are changed to the AD type. Hence, concurrent usage of these distinct types was neither expected nor supported. In Figure A.3, an example of a wrapped send operation is shown. The `AMPI_DOUBLE` datatype is defined in the library and is an alias for `MPI_DOUBLE`. This causes erroneous communication, as AdjointMPI can not make

the distinction between active and passive communication w.r.t. the double data type. Subsequently, the code was changed to accommodate the concurrent existence of these types in a program. This resulted in internal changes to almost all adjoint MPI routines of the AMPI library.

```
1  routine AMPI_Send:
2    In: buffer B, MPI_Datatype T
3    if T is not equal AMPI_DOUBLE:
4      MPI_Send(B as T);
5    else:
6      B̃ = primal_values_of(B);
7      AMPI_tape_handling(B);
8      AMPI_Send_f(B̃ as T);
```

```
routine AMPI_Send:
  In: buffer B, MPI_Datatype T
  if T is not equal AMPI_ADOUBLE:
    MPI_Send(B as T);
  else:
    B̃ = primal_values_of(B);
    AMPI_tape_handling(B);
    AMPI_Send_f(B̃ as MPI_DOUBLE);
```

Figure A.3: Pseudo code for AdjointMPI send. Original on the left: For every `AMPI_DOUBLE` datatype, the internal adjoint operations are executed (line 6–8). Otherwise, a passive send is executed (line 3–4). On the right: A new active MPI datatype, distinct from `MPI_DOUBLE`, is introduced (line 3). The adjoint send operation has a fixed type now (line 8).

Code defects The bugs in AdjointMPI were related to the internal adjoint book keeping of a subset of the aforementioned four collective routines. Errors were, for instance, (1) related to operations that should only be executed on the `root` process instead of all processes as was the case, (2) heap memory allocations with the wrong size argument, or (3) missing increments of an index variable used to access a receive buffer.

A.2.2 Improving memory management overhead in ISSM/ADOL-C

Initial performance measurements revealed high overheads for finer mesh resolutions with ADOL-C. Profiling the original ISSM implementation identified the search for contiguous memory for the active type arrays as the root cause. To remedy this, a flag was added by the ADOL-C developers to the allocation function `xNew` to specify whether contiguous memory locations are required, see [59]. Overall, 26 changes in the source code spread over 8 files for reintegrating non-contiguous locations were made. Speedup factors caused by this change are approximately 1 to 28 depending on the mesh resolution (higher resolution equals higher speedup).

Evaluation of the improvements

The comparative timings of this section between the two ADOL-C utilizations in ISSM were not presented previously.

Modeling of ice Modeling ice is a complex multi-parameter problem involving ice dynamics but also the interaction between earth systems. The behaviour of ice is governed by a system of partial differential equations known as Full-Stokes which assume the incompressibility of ice. While ISSM is capable of modeling ice using Full-Stokes, often simplified equations are employed to decrease the overall computational overhead and enable the use of a finer mesh. The evaluation is based on the Shallow-Shelf-Approximation formulation which is a 2D model derived from the Full-Stokes 3D model equations by assuming, e.g., the absence of bridging effects [133]. For more details on ice modeling see [100].

Evaluation The evaluations were conducted on compute nodes of the Lichtenberg high-performance computer of TU Darmstadt. Each node has two Intel Xeon Processor E5–2680 v3 set to a fixed frequency of 2.5 GHz with 64 GB RAM. ADOL-C was configured to have large enough buffers to avoid tape related file IO.

The ISSM execution can be logically divided into three phases: (1) Initialization, (2) forward evaluation (core), and (3) adjoint computation (AD core). However, the initialization phase has almost identical timing measurements for the two ADOL-C variants and is, thus, omitted for readability. The timings for the phases are collected by separating them with an MPI barrier:

1. **Core** The model computation, including the invocation of the linear system solver MUMPS.
2. **AD Core** Tracing is deactivated and the derivatives of the *Core* trace are generated by calling the necessary AD driver functions.

Impact of contiguous memory management To quantify the effect of the search for contiguous memory and the subsequent improvements that were made by an AD expert, the runtime of a model from the ISSM test suite is compared. The model is based on the modified test 3015 [84] and uses MUMPS. The test computes the ice volume gradient w.r.t. thickness on a square domain mesh. It was configured to run on 16 MPI processes distributed on a single compute node with varying mesh resolutions, see Figure A.4.

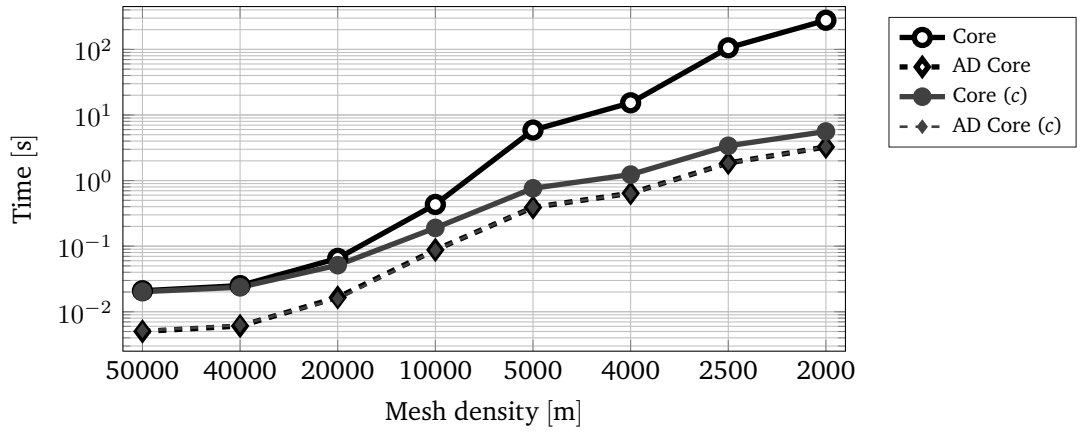


Figure A.4: Effect of the search for contiguous memory: Timings for 16 MPI processes w.r.t. different mesh densities in m (mean distance between two mesh nodes). The conditional contiguous allocation variant is indicated by (*c*).

With a finer mesh resolution, the runtime differences are apparent for the core computation. In contrast, the AD core and the init phase (not shown) have almost no difference between the two tested versions. If the total runtime is compared, a speedup of about 28 was achieved for the highest mesh resolution.

A.3 AD-enhancement of LULESH 2.0

The type change of LULESH 2.0 is straightforward. The code uses a global alias for the scalar floating-point values. Hence, it was redefined to use the CoDiPack RM overloading type. The MPI routines were redefined using MeDiPack, which is achieved by adding the `AMPI` prefix, replacing `MPI`, for each routine.

A.3.1 Main compute loop

The main compute loop with the AD related routines is shown in Listing A.5. The routines with the `AD` prefix are a thin wrapper for the CoDiPack API.

```
1  int main(int argc, char *argv[]) {
2    Domain* locDom; // problem domain with all relevant fields
3    ...
4    // Main driving loop:
5    while((locDom->time()<locDom->stoptime())&&(locDom->cycle()<opts.its)){
6      TimeIncrement(*locDom) ;
7      AD_start(); // start tracing
8      auto numElemReg = locDom->numElem();
9      for (Index_t i=0; i<numElemReg; ++i) {
10       AD_indep(locDom->p(i), ADField::p); // declare independent
11     }
12     LagrangeLeapFrog(*locDom);
13     if(myRank==0){
14       AD_dep(locDom->e(0), ADField::e); // declare dependent
15     }
16     AD_end(); // stop tracing
17     AD_driver(locDom->e(0)); // compute derivative of e(0) w.r.t. p(1:n)
18     AD_reset(); // reset the tape
19     ...
20   }
21   ...
22 } // end main
```

Listing A.5: The main loop applies a time-stepping leap frog scheme to a problem domain `locDom`. The derivative of the energy (`Domain::e`) at the origin of the domain is computed w.r.t. the pressure `Domain::p`.

A.3.2 Scalar type change

In Listing A.6, the required additional code for the AD type change is shown. All of these declarations and routines are defined in a centralized header, which is included in all translation units.

```
#include <codi.hpp>
using adreal = codi::RealReverse;
typedef adreal Real_t;  // floating point representation
...
// added overloads for AD
inline adreal SQRT(adreal arg) { return sqrt(arg); }
inline adreal FABS(adreal arg) { return abs(arg); }
inline adreal CBRT(adreal arg) { return cbrt(arg); }
// added overloads for C routines
template<typename ... Args>
void printf_oo(const char *fmt, Args &&... args) {...}
template<typename ... Args>
void fprintf_oo(FILE *stream, const char *fmt, Args &&... args) {...}
template<typename ... Args>
void sprintf_oo(char *stream, const char *fmt, Args &&... args) {...}
```

Listing A.6: Required definitions and overloads for AD overloading.

A.3.3 Adjoint MPI-related changes

In Listing A.7, the definitions and overloads for the adjoint MPI library MeDiPack are shown. The `ampi_datatype` routine is an abstraction to pass the correct datatype for the MPI call. Again, these are part of the centralized header.

The use is exemplified for an excerpt of the communication routine of the LULESH code in Listing A.8. Here, the code changes are minimal and pertain to changes to the prefix of the signature. The AD-related book-keeping is taken care of internally by the adjoint library itself.

```
#include <mpi.h>
#include <medi/medi.hpp>
#include <codi/externals/codiMediPackTypes.hpp>
using namespace medi;
using adtool = CoDiPackTool<codi::RealReverse>;
namespace detail_mpi {
  template<typename T> struct ForSpecialization {
    static auto mpi_datatype() { return MPI_DOUBLE; }
  };
  template<> struct ForSpecialization<float> {
    static auto mpi_datatype() { return MPI_FLOAT; }
  };
  template<typename Tape>
  struct ForSpecialization<codi::ActiveReal<Tape>> {
    static auto mpi_datatype() { return adtool::MPI_TYPE; }
  };
} /* namespace detail */
template<typename T> auto ampi_datatype() {
  return detail_mpi::ForSpecialization<T>::mpi_datatype();
}
```

Listing A.7: Required definitions and overloads for the adjoint MPI library.

```
void CommRecv(Domain& domain, int msgType, Index_t xferFields,
              Index_t dx, Index_t dy, Index_t dz,
              bool doRecv, bool planeOnly) {
  auto baseType = ampi_datatype<Real_t>();
  ...
  AMPI_Comm_rank(AMPI_COMM_WORLD, &myRank);
  /* post receives */
  /* receive data from neighboring domain faces */
  if (planeMin && doRecv) {
    int fromRank = myRank - domain.tp()*domain.tp();
    int recvCount = dx * dy * xferFields;
    AMPI_Irecv(&domain.commDataRecv[pmsg * maxPlaneComm],
               recvCount, baseType, fromRank, msgType,
               AMPI_COMM_WORLD, &domain.recvRequest[pmsg]);
    ++pmsg;
  }
  ...
}
```

Listing A.8: Modified MPI communication routine in `lulesh-comm.cc`.

A.4 Finding minimal cuts in the graph

```
1  // Meta data to store path, and visits
2  struct NodeData {
3    ID last_succ{};
4    bool inverted{false};
5    ID last_pred{};
6    int vis_fw{0}, vis_bw{0};
7  };
8  void MinCut::explore() {
9    // add a helper node with an edge to all input nodes:
10   ID source = add_virtual_source(graph);
11   // find paths through the graph:
12   nodeData[source].inverted = true;
13   do {
14     flow_count++;
15     nodeData[source].last_succ.reset();
16     bool has_path = visitBackward(source, {});
17   } while (has_path);
18   // Determine the final narrow cut:
19   for (const auto& [id, node] : graph->getNodes()) {
20     NodeData& data = nodeData[id];
21     if (data.inverted && (data.vis_fw == flow_count || data.vis_bw ==
         flow_count)) {
22       // node is reachable from input
23       for (ID succ : graph->successors(id)) {
24         NodeData& succ_data = nodeData[succ];
25         if (succ_data.vis_fw != flow_count && succ_data.vis_bw !=
             flow_count) {
26           // successor is not reachable from input
27           node_cut.push_back(id);
28           break;
29         }
30       }
31     }
32   }
33   // Remove previously added single source node
34   graph->removeNode(source);
35 }
```

Listing A.9: Main driver for the cut algorithm. The algorithm tries to find all unique paths to a sink from a virtual source. Once this is not possible, meta information are used to determine the narrow point.

```
1  bool MinCut::visitForward(ID node, ID from) {
2    // Get node data, avoid loops, update forward edge visit count
3    NodeData& data = nodeData[node];
4    if (data.vis_fw == flow_count) {
5      return false;
6    }
7    data.vis_fw = flow_count;
8    // Is the node part of an existing path?:
9    if (data.inverted) {
10     // We continue the search from the predecessor of the node
11     bool new_path = visitBackward(data.last_pred.value(), node);
12     if (new_path) {
13       data.last_pred = from;
14       return true;
15     }
16   } else {
17     // Reached a sink?
18     if (graph->successors(node).empty()) {
19       data = NodeData{from, true, {}, data.vis_fw, data.vis_bw};
20       return true;
21     }
22     // Did not reach sink, build path to sink recursively:
23     for (ID succ : graph->successors(node)) {
24       bool has_path = visitForward(succ, node);
25       if (has_path) {
26         data = NodeData{from, true, succ, data.vis_fw, data.vis_bw};
27         return true;
28       }
29     }
30   }
31   return false;
32 }
```

Listing A.10: Forward depth-first search to build a path to a sink, handles nodes of existing paths by following the back-edge.

```
1  bool MinCut::visitBackward(ID node, std::optional<ID> from) {
2    // Get node data, avoid loops, update backward edge visit count
3    NodeData& data = nodeData[node];
4    if (data.vis_bw == flow_count) {
5      return false;
6    }
7    data.vis_bw = flow_count;
8    // Depth first search, follow successors recursively:
9    for (ID succ : graph->successors(node)) {
10     if (succ == from) { continue; }
11     bool has_path = visitForward(succ, node);
12     if (has_path) {
13       data.last_succ = succ;
14       return true;
15     }
16   }
17   // Follow path backward?:
18   if (!graph->predecessors(node).empty()) {
19     bool bsearch = visitBackward(data.last_pred.value(), node);
20     if(bsearch) {
21       // We followed the path, reset the node to remove from any path.
22       data = NodeData{{}, false, {}, data.vis_fw, data.vis_bw};
23       return true;
24     }
25   }
26   return false;
27 }
```

Listing A.11: Backward search: For each node try all edges forward or follow a back-edge and try again.

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, N. R. Tallent. **"HPCTOOLKIT: tools for performance analysis of optimized parallel programs"**. In: *Concurrency Computat.: Pract. Exper.* 22.6 (2010), pp. 685–701. DOI: 10.1002/cpe.1553.

[2] S. Akbarzadeh, J. Hückelheim, J. Müller. **"Consistent treatment of incompletely converged iterative linear solvers in reverse-mode AD"**. In: *AD 2016. The 7th International Conference on Algorithmic Differentiation - Programme and Abstracts* (2016), pp. 19–22. URL: www.autodiff.org/ad16/AD2016ProgrammeAndAbstracts.pdf.

[3] T. A. Albring, M. Sagebaum, N. R. Gauger. **"Development of a Consistent Discrete Adjoint Solver in an Evolving Aerodynamic Design Framework"**. In: *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. AIAA, 2015. DOI: 10.2514/6.2015-3240.

[4] A. Alexandrescu. **"Modern C++ design: generic programming and design patterns applied"**. Addison-Wesley, 2001. ISBN: 0-201-70431-5.

[5] P. J. Applegate, N. Kirchner, E. J. Stone, K. Keller, R. Greve. **"An assessment of key model parametric uncertainties in projections of Greenland Ice Sheet behavior"**. In: *The Cryosphere* 6.3 (2012), pp. 589–606. DOI: 10.5194/tc-6-589-2012.

[6] M. Asch, M. Bocquet, M. Nodet. **"Data Assimilation: Methods, Algorithms, and Applications"**. SIAM, 2016. DOI: 10.1137/1.9781611974546.

[7] P. Aubert, N. D. Césaré. **"Expression Templates and Forward Mode Automatic Differentiation"**. In: *Automatic Differentiation of Algorithms*. Springer, 2002, pp. 311–315. DOI: 10.1007/978-1-4613-0075-5_37.

[8] **"Auto-Vectorization in LLVM"**. Online. *Last accessed* Jan 2020. URL: https://llvm.org/docs/Vectorizers.html.

[9] B. M. Averick, R. G. Carter, G.-L. Xue, J. Moré. **"The MINPACK-2 test problem collection"**. Tech. rep. Argonne National Lab., IL (United States), 1992.

[10] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi. **"A Survey of Symbolic Execution Techniques"**. In: *ACM Comput. Surv.* 51.3 (May 2018). doi: 10.1145/3182657.

[11] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, M. V. Zelkowitz. **"Understanding the High-Performance-Computing Community: A Software Engineer's Perspective"**. In: *IEEE* 25.4 (July 2008), pp. 29–36. doi: 10.1109/MS.2008.103.

[12] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, N. Wang. **"XRay: A Function Call Tracing System"**. Tech. rep. 2016. url: https://ai.google/research/pubs/pub45287.

[13] C. H. Bischof, H. Bücker, B. Lang, A. Rasch, A. Vehreschild. **"Combining Source Transformation and Operator Overloading Techniques to Compute Derivatives for MATLAB Programs"**. In: *2nd IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2002, pp. 65–72. doi: 10.1109/SCAM.2002.1134106.

[14] C. H. Bischof, M. R. Haghighat. **"Hierarchical Approaches to Automatic Differentiation"**. In: *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, 1996, pp. 83–94.

[15] C. H. Bischof, A. Carle, P. Khademi, A. Mauer. **"ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs"**. In: *IEEE Computational Science & Engineering* 3.3 (1996), pp. 18–32.

[16] Y. Boykov, V. Kolmogorov. **"An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision"**. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.9 (Sept. 2004), pp. 1124–1137. doi: 10.1109/TPAMI.2004.60.

[17] I. N. Bronstein, K. A. Semendyayev. **"Handbook of Mathematics"**. Springer, 2007. doi: 10.1007/978-3-540-72122-2.

[18] H. M. Bücker. **"Looking for narrow interfaces in automatic differentiation using graph drawing"**. In: *Future Generation Computer Systems* 21.8 (2005), pp. 1418–1425. doi: 10.1016/j.future.2004.11.007.

[19] H. M. Bücker, A. Rasch. **"Modeling the performance of interface contraction"**. In: *ACM Trans. Math. Software* 29.4 (2003), pp. 440–457. doi: 10.1145/962437.962442.

[20] C. Cadar, D. Dunbar, D. R. Engler, et al. **"KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs."** In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[21] G. R. Carmichael, A. Sandu, F. A. Potra. **"Sensitivity Analysis For Atmospheric Chemistry Models Via Automatic Differentiation"**. In: *Atmos. Environ.* 31.3 (1997), pp. 475–489. doi: 10.1016/S1352-2310(96)00168-9.

[22] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald. **"Parallel programming in OpenMP"**. Morgan Kaufmann, 2001. isbn: 978-1-55860-671-5.

[23] S. Chunduri, S. Parker, P. Balaji, K. Harms, K. Kumaran. **"Characterization of MPI Usage on a Production Supercomputer"**. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. IEEE, 2018, 30:1–30:15. doi: 10.1109/SC.2018.00033.

[24] **"CORAL benchmark codes"**. Online. *Last accessed* Jan 2020. url: https://asc.llnl.gov/CORAL-benchmarks/.

[25] F. Devernay. **"C/C++ Minpack"**. Online. *Last accessed* Jan 2020. 2007. url: http://devernay.free.fr/hacks/cminpack.

[26] J. Dongarra, A. D. Malony, S. Moore, P. Mucci, S. Shende. **"Performance Instrumentation and Measurement for Terascale Systems"**. In: *Computational Science — ICCS 2003*. Springer, 2003, pp. 53–62. doi: 10.1007/3-540-44864-0_6.

[27] C. C. Douglas, K. Krishnamoorthy. **"Static Analysis and Symbolic Execution for Deadlock Detection in MPI Programs"**. In: *Computational Science – ICCS 2018*. Springer, 2018, pp. 783–796. doi: 10.1007/978-3-319-93701-4_62.

[28] A. Droste, M. Kuhn, T. Ludwig. **"MPI-Checker: Static Analysis for MPI"**. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: ACM, 2015, 3:1–3:10. doi: 10.1145/2833157.2833159.

[29] G. J. Duck, R. H. C. Yap. **"EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++"**. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. ACM, 2018, pp. 181–195. doi: 10.1145/3192366.3192388.

[30] T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, J. J. Alonso. **"SU2: An Open-Source Suite for Multiphysics Simulation and Design"**. In: *AIAA Journal* 54.3 (2016), pp. 828–846. doi: 10.2514/1.J053813.

[31] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, G. Woodhull. **"Graphviz – Open Source Graph Drawing Tools"**. In: *International Symposium on Graph Drawing*. Springer. 2001, pp. 483–484. DOI: 10.1007/3-540-45848-4_57.

[32] M. Fagan, L. Hascoet, J. Utke. **"Data Representation Alternatives in Semantically Augmented Numerical Models"**. In: *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. Sept. 2006, pp. 85–94. DOI: 10.1109/SCAM.2006.11.

[33] H. Falk, P. Marwedel. **"Control Flow Driven Splitting of Loop Nests at the Source Code Level"**. In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*. IEEE, 2003, p. 10410. DOI: 10.1109/DATE.2003.1253644.

[34] C. Faure, U. Naumann. **"Minimizing the Tape Size"**. In: *Automatic Differentiation of Algorithms*. Springer, 2002, pp. 293–298. DOI: 10.1007/978-1-4613-0075-5_34.

[35] L. R. Ford, D. R. Fulkerson. **"Maximal Flow Through A Network"**. In: *Classic Papers in Combinatorics*. Springer, 2009, pp. 243–248. DOI: 10.1007/978-0-8176-4842-8_15.

[36] M. Franke, T. Röber, E. Kügeler, G. Ashcroft. **"Turbulence treatment in steady and unsteady turbomachinery flows"**. In: *V European conference on computational fluid dynamics, ECCOMAS CFD*. 2010, pp. 14–17. ISBN: 978-989-96778-1-4.

[37] R. Giering, T. Kaminski. **"Automatic Sparsity Detection Implemented as a Source-to-Source Transformation"**. In: *Computational Science – ICCS 2006*. Springer, 2006, pp. 591–598. DOI: 10.1007/11758549_81.

[38] R. Giering, T. Kaminski. **"Recipes for Adjoint Code Construction"**. In: *ACM Trans. Math. Software* 24.4 (1998), pp. 437–474. DOI: 10.1145/293686.293695.

[39] M. B. Giles. **"Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation"**. In: *Advances in Automatic Differentiation*. Springer, 2008, pp. 35–44. DOI: 10.1007/978-3-540-68942-3_4.

[40] L. Glenn, C. Hua, C. James, H. Jim, K. Marina, Z. Yan. **"MPI-CHECK: a tool for checking Fortran 90 MPI programs"**. In: *Concurrency and Computation: Practice and Experience* 15.2 (2003), pp. 93–100.

[41] **"GNU Compiler Collection: Instrumentation Options"**. Online. *Last accessed* Jan 2020. 2019. URL: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html.

[42] H. Goelzer, P. Huybrechts, J. J. Fürst, F. Nick, M. L. Andersen, T. L. Edwards, X. Fettweis, A. J. Payne, S. Shannon. **"Sensitivity of Greenland ice sheet projections to model formulations"**. In: *J. Glaciol.* 59.216 (2013), pp. 733–749. doi: 10.3189/2013JoG12J182.

[43] B. Gregg. **"The Flame Graph"**. In: *Commun. ACM* 59.6 (May 2016), pp. 48–57. doi: 10.1145/2909476.

[44] B. Gregg. **"Thinking Methodically About Performance"**. In: *Queue* 10.12 (Dec. 2012), 40:40–40:51. doi: 10.1145/2405116.2413037.

[45] A. Griewank, A. Walther. **"Evaluating Derivatives"**. Second. SIAM, 2008. doi: 10.1137/1.9780898717761.

[46] A. Griewank, D. Juedes, J. Utke. **"Algorithm 755: ADOL-C: A Package For The Automatic Differentiation Of Algorithms Written In C/C++"**. In: *ACM Trans. Math. Software* 22.2 (1996), pp. 131–167. doi: 10.1145/229473.229474.

[47] A. Griewank, A. Walther. **"Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation"**. In: *ACM Trans. Math. Software* 26.1 (Mar. 2000), pp. 19–45. doi: 10.1145/347837.347846.

[48] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, E. van der Kouwe. **"TypeSan: Practical Type Confusion Detection"**. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. ACM, 2016, pp. 517–528. doi: 10.1145/2976749.2978405.

[49] L. Hascoet, V. Pascual. **"The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification"**. In: *ACM Trans. Math. Softw.* 39.3 (May 2013), 20:1–20:43. doi: 10.1145/2450153.2450158.

[50] L. Hascoët, M. Morlighem. **"Source-to-source adjoint Algorithmic Differentiation of an ice sheet model written in C"**. In: *Optim. Method. Softw.* 33.4-6 (2018), pp. 829–843. doi: 10.1080/10556788.2017.1396600.

[51] L. Hascoët, B. Dauvergne. **"Adjoints of large simulation codes through Automatic Differentiation"**. In: *European Journal of Computational Mechanics* 17.1-2 (2008), pp. 63–86. doi: 10.3166/remn.17.63-86.

[52] L. Hascoët, S. Fidanova, C. Held. **"Adjoining Independent Computations"**. In: *Automatic Differentiation of Algorithms*. Springer, 2002, pp. 299–304. doi: 10.1007/978-1-4613-0075-5_35.

[53] J. He, A. E. Snavely, R. F. V. d. Wijngaart, M. A. Frumkin. **"Automatic Recognition of Performance Idioms in Scientific Applications"**. In: *2011 IEEE International Parallel Distributed Processing Symposium*. May 2011, pp. 118–127. doi: 10.1109/IPDPS.2011.21.

[54] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, M. S. Müller. **"MPI runtime error detection with MUST: Advances in deadlock detection"**. In: *Scientific Programming* 21.3-4 (2013), pp. 109–121. doi: 10.1109/SC.2012.79.

[55] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, C. S. Woodward. **"SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers"**. In: *ACM Trans. Math. Software* 31.3 (2005), pp. 363–396. doi: 10.1145/1089014.1089020.

[56] R. J. Hogan. **"Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++"**. In: *ACM Trans. Math. Software* 40.4 (July 2014), 26:1–26:16. doi: 10.1145/2560359.

[57] J. Howison, E. Deelman, M. J. McLennan, R. Ferreira da Silva, J. D. Herbsleb. **"Understanding the scientific software ecosystem and its impact: Current and future measures"**. In: *Research Evaluation* 24.4 (July 2015), pp. 454–470. doi: 10.1093/reseval/rvv014.

[58] A. Hück, C. Bischof. **"Operator Overloading Compatibility for AD - A Case Study of Scientific C++ Codes"**. In: *AD 2016. The 7th International Conference on Algorithmic Differentiation - Programme and Abstracts* (2016), pp. 87–90. url: www.autodiff.org/ad16/AD2016ProgrammeAndAbstracts.pdf.

[59] A. Hück, C. Bischof, M. Sagebaum, N. R. Gauger, B. Jurgelucks, E. Larour, G. Perez. **"A Usability Case Study of Algorithmic Differentiation Tools on the ISSM Ice Sheet Model"**. In: *Optim. Method. Softw.* 33.4–6 (2018), pp. 844–867. doi: 10.1080/10556788.2017.1396602.

[60] A. Hück, C. Bischof, J. Utke. **"Checking C++ Codes for Compatibility with Operator Overloading"**. In: *15th IEEE International Working Conference on Source Code Analysis and Manipulation*. Vol. 15. IEEE, 2015, pp. 91–100. doi: 10.1109/SCAM.2015.7335405.

[61] A. Hück, S. Kreutzer, D. Messig, A. Scholtissek, C. Bischof, C. Hasse. **"Application of Algorithmic Differentiation for Exact Jacobians to the Universal Laminar Flame Solver"**. In: *Computational Science - ICCS 2018*. Vol. 10862. Lecture Notes in Computer Science. Springer, 2018, pp. 480–486. doi: 10.1007/978-3-319-93713-7_43.

[62] A. Hück, J.-P. Lehr, S. Kreutzer, J. Protze, C. Terboven, C. Bischof, M. S. Müller. **"Compiler-aided type tracking for correctness checking of MPI applications"**. In: *2nd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, Nov. 2018, pp. 51–58. doi: 10.1109/Correctness.2018.00011.

[63] A. Hück, J. Utke, C. Bischof. **"Source Transformation of C++ Codes for Compatibility with Operator Overloading"**. In: *Procedia Comput. Sci.* 80 (2016), pp. 1485–1496. doi: 10.1016/j.procs.2016.05.470.

[64] R. Hyde. **"The Fallacy of Premature Optimization"**. In: *Ubiquity* 2009.February (2009). doi: 10.1145/1569886.1513451.

[65] **"ISO/IEC 14882:1998"**. C++. International Standards Organisation (ISO). 1998. 732 pp. url: www.iso.org/standard/25845.html.

[66] **"ISO/IEC 14882:2003"**. C++. International Standards Organisation (ISO). 2003. 757 pp. url: www.iso.org/standard/38110.html.

[67] **"ISO/IEC 14882:2011"**. C++. International Standards Organisation (ISO). 2011. 1338 pp. url: www.iso.org/standard/50372.html.

[68] **"ISO/IEC 14882:2014"**. C++. International Standards Organisation (ISO). 2014. 1358 pp. url: www.iso.org/standard/64029.html.

[69] **"ISO/IEC 14882:2017"**. C++. International Standards Organisation (ISO). 2017. 1605 pp. url: www.iso.org/standard/68564.html.

[70] C. Iwainsky. **"InstRO: a component-based toolbox for performance instrumentation"**. PhD thesis, Technische Universität Darmstadt, 2015. PhD thesis. Aachen: TU Darmstadt, 2016. isbn: 978-3-8440-4562-8.

[71] W. Jakob. **"Enoki: structured vectorization and differentiation on modern processor architectures"**. Online. *Last accessed* Jan 2020. 2019. url: https://github.com/mitsuba-renderer/enoki.

[72] H. Jasak, A. Jemcov, Z. Tukovic, et al. **"OpenFOAM: A C++ library for complex physics simulations"**. In: *International Workshop on Coupled Methods in Numerical Dynamics*. Vol. 1000. IUC Dubrovnik Croatia. 2007, pp. 1–20.

[73] Y. Jeon, P. Biswas, S. Carr, B. Lee, M. Payer. **"HexType: Efficient Detection of Type Confusion Errors for C++"**. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. ACM, 2017, pp. 2373–2387. doi: 10.1145/3133956.3134062.

[74] M. E. Jerrel. **"Automatic Differentiation and Interval Arithmetic for Estimation of Disequilibrium Models"**. In: *Computational Economics* 10.3 (Aug. 1997), pp. 295–316. doi: 10.1023/A:1008633613243.

[75] D. Kelly. **"Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software"**. In: *Journal of Systems and Software* 109 (2015), pp. 50–61. doi: 10.1016/j.jss.2015.07.027.

[76] D. E. Keyes, L. C. McInnes, C. Woodward, et al. **"Multiphysics simulations: Challenges and opportunities"**. In: 27.1 (2013), pp. 4–83. doi: 10.1177/1094342012468181.

[77] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, F. Wolf. **"Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir"**. In: *Tools for High Performance Computing 2011*. Springer, 2012. doi: 10.1007/978-3-642-31476-6_7.

[78] A. Koniges, B. Cook, J. Deslippe, T. Kurth, H. Shan. **"MPI Usage at NERSC: Present and Future"**. In: *Proceedings of the 23rd European MPI Users' Group Meeting*. EuroMPI 2016. ACM, 2016, pp. 217–217. doi: 10.1145/2966884.2966894.

[79] B. Krammer, K. Bidmon, M. Müller, M. Resch. **"MARMOT: An MPI analysis and checking tool"**. In: *Parallel Computing*. Vol. 13. Advances in Parallel Computing. North-Holland, 2004, pp. 493–500. doi: 10.1016/S0927-5452(04)80063-7.

[80] K. Kulshreshtha, S. H. K. Narayanan, T. Albring. **"A Mixed Approach to Adjoint Computation with Algorithmic Differentiation"**. In: *System Modeling and Optimization*. Springer, 2016, pp. 331–340. doi: 10.1007/978-3-319-55795-3_31.

[81] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, N. Sultana. **"A Large-scale Study of MPI Usage in Open-source HPC Applications"**. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. ACM, 2019, 31:1–31:14. doi: 10.1145/3295500.3356176.

[82] W. Landi. **"Undecidability of static analysis"**. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337. DOI: 10.1145/161494.161501.

[83] E. Larour, H. Seroussi, M. Morlighem, E. Rignot. **"Continental scale, high order, high spatial resolution, ice sheet modeling using the Ice Sheet System Model (ISSM)"**. In: *Journal of Geophysical Research: Earth Surface* 117.F1 (2012), pp. 1–20. DOI: 10.1029/2011JF002140.

[84] E. Larour, J. Utke, A. Bovin, M. Morlighem, G. Perez. **"An approach to computing discrete adjoints for MPI-parallelized models applied to Ice Sheet System Model 4.11"**. In: *Geosci. Model Dev.* 9.11 (2016), pp. 3907–3918. DOI: 10.5194/gmd-9-3907-2016.

[85] C. Lattner, V. Adve. **"LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation"**. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. 2004. ISBN: 0-7695-2102-9.

[86] B. Lee, C. Song, T. Kim, W. Lee. **"Type Casting Verification: Stopping an Emerging Attack Vector"**. In: *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015, pp. 81–96. ISBN: 978-1-931971-232.

[87] J.-P. Lehr, A. Hück, C. Bischof. **"PIRA: Performance Instrumentation Refinement Automation"**. In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*. AI-SEPS 2018. ACM, 2018, pp. 1–10. DOI: 10.1145/3281070.3281071.

[88] K. Leppkes, J. Lotz, U. Naumann, J. du Toit. **"Meta Adjoint Programming in C++"**. Tech. rep. RWTH Aachen, AIB-2017-07, 2017.

[89] **"LLVM Loop Terminology (and Canonical Forms)"**. Online. *Last accessed* Jan 2020. URL: https://llvm.org/docs/LoopTerminology.html.

[90] J. Lotz. **"Hybrid approaches to adjoint code generation with dco/c++"**. PhD thesis. Aachen: RWTH Aachen University, 2016, p. 127. URL: https://publications.rwth-aachen.de/record/667318.

[91] J. Lotz, U. Naumann, S. Mitra. **"Mixed Integer Programming for Call Tree Reversal"**. In: *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, pp. 83–91. DOI: 10.1137/1.9781611974690.ch9.

[92] J. Lotz, U. Naumann, J. Ungermann. **"Hierarchical Algorithmic Differentiation: A Case Study"**. In: *Recent Advances in Algorithmic Differentiation*. Springer, 2012, pp. 187–196. DOI: 10.1007/978-3-642-30023-3_17.

[93] B. Lu, J. Mellor-Crummey. **"Compiler optimization of implicit reductions for distributed memory multiprocessors"**. In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. Mar. 1998, pp. 42–51. DOI: 10.1109/IPPS.1998.669887.

[94] **"LULESH 2.0 Updates and Changes"**. Tech. rep. Lawrence Livermore National Lab (LLNL). URL: https://computing.llnl.gov/projects/co-design/lulesh2.0_changes1.pdf.

[95] M. Martonosi, A. Gupta, T. Anderson. **"Effectiveness of Trace Sampling for Performance Debugging Tools"**. In: *SIGMETRICS Perform. Eval. Rev.* 21.1 (June 1993), pp. 248–259. DOI: 10.1145/166962.167023.

[96] H. Menon, M. Lam, D. Kuffour, M. Schordan, S. Llyod, K. Mohror, J. Hittinger. **"ADAPT: Algorithmic Differentiation for Floating-Point Precision Tuning"**. Tech. rep. Lawrence Livermore National Lab (LLNL), 2018. URL: www.osti.gov/servlets/purl/1488804.

[97] Message Passing Interface Forum. **"MPI: A Message-Passing Interface Standard, Version 2.2"**. Online. *Last accessed* Jan 2020. 2009. URL: www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[98] Message Passing Interface Forum. **"MPI: A Message-Passing Interface Standard, Version 3.1"**. Online. *Last accessed* Jan 2020. 2015. URL: www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[99] L. Moody, N. Pinnow, M. O. Lam, H. Menon, M. Schordan, G. S. Lloyd, T. Islam. **"Automatic Generation of Mixed-Precision Programs: Extended Abstract"**. In: *Proceedings of Supercomputing (SC'18)*. ACM, 2018, p. 2. URL: https://sc18.supercomputing.org/proceedings/tech_poster/poster_files/post219s2-file3.pdf.

[100] M. Morlighem. **"Ice sheet properties inferred by combining numerical modeling and remote sensing data"**. PhD thesis. Ecole Centrale Paris, Dec. 2011. URL: https://tel.archives-ouvertes.fr/tel-00697004.

[101] M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, C. Ponder. **"SPEC MPI2007 — an application benchmark suite for parallel systems using MPI"**. In: *Concurrency and Computation: Practice and Experience* 22.2 (2009), pp. 191–205. DOI: 10.1002/cpe.1535.

[102] S. H. K. Narayanan, B. Norris, B. Winnicka. **"ADIC2: Development of a component source transformation system for differentiating C and C++"**. In: *Procedia Comput. Sci.* 1.1 (2010), pp. 1845–1853. DOI: 10.1016/j.procs.2010.04.206.

[103] U. Naumann. **"Call Tree Reversal is NP-Complete"**. In: *Advances in Automatic Differentiation*. Springer, 2008, pp. 13–22. DOI: 10.1007/978-3-540-68942-3_2.

[104] U. Naumann. **"The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation"**. Vol. 1. SIAM, 2012. DOI: 10.1137/1.9781611972078.

[105] U. Naumann, J. Lotz, K. Leppkes, M. Towara. **"Algorithmic Differentiation of Numerical Methods: Tangent and Adjoint Solvers for Parameterized Systems of Nonlinear Equations"**. In: *ACM Trans. Math. Software* 41.4 (Oct. 2015), 26:1–26:21. DOI: 10.1145/2700820.

[106] U. Naumann, J. Toit. **"Adjoint Algorithmic Differentiation Tool Support for Typical Numerical Patterns in Computational Finance"**. In: *Journal of Computational Finance* 21.4 (2018). URL: https://ssrn.com/abstract=3122293.

[107] **"OProfile"**. Online. *Last accessed* Jan 2020. URL: https://oprofile.sourceforge.io.

[108] R. P. Pawlowski, E. T. Phipps, A. G. Salinger. **"Automating Embedded Analysis Capabilities and Managing Software Complexity in Multiphysics Simulation, Part I: Template-based Generic Programming"**. In: *Sci. Program.* 20.2 (2012), pp. 197–219. DOI: 10.3233/SPR-2012-0350.

[109] **"perf: Linux profiling with performance counters"**. Online. *Last accessed* Jan 2020. URL: https://perf.wiki.kernel.org/.

[110] E. Phipps, D. Gay. **"Sacado: Automatic Differentation Tools for C++ Applications"**. Online. *Last accessed* Jan 2020. URL: https://trilinos.github.io/sacado.html.

[111] E. Phipps, R. Pawlowski. **"Efficient Expression Templates for Operator Overloading-Based Automatic Differentiation"**. In: *Recent Advances in Algorithmic Differentiation*. Springer, 2012, pp. 309–319. DOI: 10.1007/978-3-642-30023-3_28.

[112] E. T. Phipps, R. A. Bartlett, D. M. Gay, R. J. Hoekstra. **"Large-Scale Transient Sensitivity Analysis of a Radiation-Damaged Bipolar Junction Transistor via Automatic Differentiation"**. In: *Advances in Automatic Differentiation*. Springer, 2008, pp. 351–362. DOI: 10.1007/978-3-540-68942-3_31.

[113] D. E. Post, R. P. Kendall, E. M. Whitney. **"Case Study of the Falcon Code Project"**. In: *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*. SE-HPCS '05. ACM, 2005, pp. 22–26. DOI: 10.1145/1145319.1145327.

[114] M. Priestley. **"The Logic of Correctness in Software Engineering"**. In: *A Science of Operations: Machines, Logic and the Invention of Programming*. London: Springer, 2011, pp. 253–276. DOI: 10.1007/978-1-84882-555-0_10.

[115] M. Roughan, J. Tuke. **"The Hitchhikers Guide to Sharing Graph Data"**. In: *2015 3rd International Conference on Future Internet of Things and Cloud*. IEEE. 2015, pp. 435–442. DOI: 10.1109/FiCloud.2015.76.

[116] N. Safiran, U. Naumann. **"Toward Adjoint OpenFOAM"**. Tech. rep. RWTH Aachen, 2011. URL: http://publications.rwth-aachen.de/record/47824.

[117] M. Sagebaum. **"Advanced techniques for the semi automatic transition from simulation to design software"**. PhD thesis. Kaiserslautern: Technische Universität Kaiserslautern, 2018, p. 211. URL: http://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-53891.

[118] M. Sagebaum, T. Albring, N. R. Gauger. **"High-Performance Derivative Computations Using CoDiPack"**. In: *ACM Trans. Math. Softw.* 45.4 (2019). DOI: 10.1145/3356900.

[119] M. Sagebaum, N. R. Gauger. **"MeDiPack – Message Differentiation Package"**. Online, *last visited Aug 2019*. Technische Universität Kaiserslautern. 2019. URL: https://github.com/scicompkl/medipack.

[120] M. Sagebaum, N. R. Gauger, U. Naumann, J. Lotz, K. Leppkes. **"Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries"**. In: *Procedia Comput. Sci.* 18 (2013), pp. 208–217. DOI: 10.1016/j.procs.2013.05.184.

[121] M. Schanen, U. Naumann, L. Hascoët, J. Utke. **"Interpretative adjoints for numerical simulation codes using MPI"**. In: *Procedia Comput. Sci.* 1.1 (2010). ICCS 2010, pp. 1825–1833. DOI: 10.1016/j.procs.2010.04.204.

[122] B. Smaalders. **"Performance Anti-Patterns"**. In: *Queue* 4.1 (Feb. 2006), pp. 44–50. DOI: 10.1145/1117389.1117403.

[123] B. Stroustrup. **"The C++ Programming Language"**. Third. Addison-Wesley, 2010. ISBN: 0-201-88954-4.

[124] B. Stroustrup. **"The C++ Programming Language"**. Fourth. Addison-Wesley, 2013. ISBN: 978-0321563842.

[125] B. Stroustrup, H. Sutter. **"C++ Core Guidelines - C.183"**. Online. *Last accessed* Jan 2020. URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c183-dont-use-a-union-for-type-punning.

[126] **"Support for contract based programming in C++"**. Online. *Last accessed* Jan 2020. URL: www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html.

[127] H. Sutter. **"A Modest Proposal: Fixing ADL (revision 2)"**. Online. *Last accessed* Jan 2020. URL: www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2103.pdf.

[128] **"The AdjoinableMPI Documentation"**. Online. *Last accessed* Jan 2020. 2019. URL: www.mcs.anl.gov/~utke/AdjoinableMPI/AdjoinableMPIDox/LibraryDevelopmentGuide.html.

[129] M. Towara, U. Naumann. **"A Discrete Adjoint Model for OpenFOAM"**. In: *Procedia Comput. Sci.* 18 (2013), pp. 429–438. DOI: 10.1016/j.procs.2013.05.206.

[130] M. Towara. **"Discrete adjoint optimization with OpenFOAM"**. PhD thesis. Aachen: RWTH Aachen University, 2018, p. 232. DOI: 10.18154/RWTH-2019-00475.

[131] J. Utke, L. Hascoet, P. Heimbach, C. Hill, P. Hovland, U. Naumann. **"Toward adjoinable MPI"**. In: *2009 IEEE International Symposium on Parallel Distributed Processing*. May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5161165.

[132] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, C. Wunsch. **"OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes"**. In: *ACM Trans. Math. Software* 34.4 (2008), p. 18. DOI: 10.1145/1377596.1377598.

[133] C. van der Veen. **"Fundamentals of Glacier Dynamics"**. Mar. 2013. DOI: 10.1201/b14059.

[134] T. Veldhuizen. **"Expression Templates"**. In: *C++ Report* 7.5 (1995), pp. 26–31. ISSN: 1040-6042.

[135] J. S. Vetter, B. R. de Supinski. **"Dynamic Software Testing of MPI Applications with Umpire"**. In: *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. SC '00. IEEE, 2000. ISBN: 0-7803-9802-5.

[136] M. Vobeck, R. Giering, T. Kaminski. **"Development and First Applications of TAC++"**. In: *Advances in Automatic Differentiation*. Springer, 2008, pp. 187–197. DOI: 10.1007/978-3-540-68942-3_17.

[137] A. Walther, A. Griewank. **"Getting started with ADOL-C"**. In: *Combinatorial Scientific Computing*. Chapman-Hall CRC, 2012. Chap. 7, pp. 181–202. DOI: 10.1201/b11644.

[138] J. Willkomm, C. Bischof, M. H. Bücker. **"RIOS: efficient I/O in reverse direction"**. In: *Softw. Pract. Exper.* 45.10 (2015), pp. 1399–1427. DOI: 10.1002/spe.2252.

[139] **"XRay Instrumentation"**. Online. *Last accessed* Jan 2020. URL: https://llvm.org/docs/XRay.html.

[140] F. Ye, J. Zhao, V. Sarkar. **"Detecting MPI Usage Anomalies via Partial Program Symbolic Execution"**. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2018, pp. 794–806. DOI: 10.1109/SC.2018.00066.

[141] B. Y. Zhou, T. Albring, N. R. Gauger, T. D. Economon, F. Palacios, J. J. Alonso. **"A Discrete Adjoint Framework for Unsteady Aerodynamic and Aeroacoustic Optimization"**. In: (2015). DOI: 10.2514/6.2015-3355.

[142] A. Zschutschke, D. Messig, A. Scholtissek, C. Hasse. **"Universal Laminar Flame Solver (ULF)"**. June 2017. DOI: 10.6084/m9.figshare.5119855.v2.

List of Abbreviations

AD	Algorithmic Differentiation
ADL	Argument-dependent Lookup
AST	Abstract Syntax Tree
CFD	Computational Fluid Dynamics
CFG	Control-Flow Graph
DAE	Differential-algebraic Equation
DAG	Directed Acyclic Graph
FD	Finite Differences
FM	Forward Mode
FPF	Freely-propagating Premixed Flame
HPC	High-Performance Computing
HR	Homogeneous Reactor
IR	Intermediate Representation
ISSM	Ice Sheet System Model
LOC	Lines of Code
MM	Mixed Mode
MPI	Message Passing Interface
ODE	Ordinary Differential Equation
overloading	Operator Overloading
RM	Reverse Mode
SAC	Single Assignment Code
TU	Translation Unit
ULF	Universal Laminar Flame

List of Figures

List of Listings

List of Tables

List of own Publications

[1] A. Hück, C. Bischof, J. Utke. **"Checking C++ Codes for Compatibility with Operator Overloading"**. In: *15th IEEE International Working Conference on Source Code Analysis and Manipulation*. Vol. 15. IEEE, 2015, pp. 91–100. DOI: 10.1109/SCAM.2015.7335405.

[2] A. Hück, J. Willkomm, C. Bischof. **"Source Transformation for the Optimized Utilization of the Matlab Runtime System for Automatic Differentiation"**. In: *Lecture Notes in Computational Science and Engineering*. Springer, 2015, pp. 115–131. DOI: 10.1007/978-3-319-22997-3_7.

[3] A. Hück, J. Utke, C. Bischof. **"Source Transformation of C++ Codes for Compatibility with Operator Overloading"**. In: *Procedia Comput. Sci.* 80 (2016), pp. 1485–1496. DOI: 10.1016/j.procs.2016.05.470.

[4] A. Hück, C. Bischof, M. Sagebaum, N. R. Gauger, B. Jurgelucks, E. Larour, G. Perez. **"A Usability Case Study of Algorithmic Differentiation Tools on the ISSM Ice Sheet Model"**. In: *Optim. Method. Softw.* 33.4–6 (2018), pp. 844–867. DOI: 10.1080/10556788.2017.1396602.

[5] A. Hück, S. Kreutzer, D. Messig, A. Scholtissek, C. Bischof, C. Hasse. **"Application of Algorithmic Differentiation for Exact Jacobians to the Universal Laminar Flame Solver"**. In: *Computational Science - ICCS 2018*. Vol. 10862. Lecture Notes in Computer Science. Springer, 2018, pp. 480–486. DOI: 10.1007/978-3-319-93713-7_43.

[6] A. Hück, J.-P. Lehr, S. Kreutzer, J. Protze, C. Terboven, C. Bischof, M. S. Müller. **"Compiler-aided type tracking for correctness checking of MPI applications"**. In: *2nd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, Nov. 2018, pp. 51–58. DOI: 10.1109/Correctness.2018.00011.

[7] J.-P. Lehr, A. Hück, C. Bischof. **"PIRA: Performance Instrumentation Refinement Automation"**. In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*. AI-SEPS 2018. ACM, 2018, pp. 1–10. DOI: 10.1145/3281070.3281071.

Academic Curriculum Vitae

Alexander Hück studied Computational Engineering at Technische Universität Darmstadt with a focus on computer science.

He acquired his bachelor degree at the Simulation, Systems Optimization and Robotics Group in 2010. After a two semester stay at Nanyang Technological University in Singapore, he received his master's degree in 2013 at the Scientific Computing Group in Darmstadt.

Subsequently, in 2014, he started his work there as a research assistant under the supervision of Prof. Dr. Christian Bischof. In 2020, he was awarded with a doctoral degree in natural sciences (Dr. rer. nat.) from Technische Universität Darmstadt. His research interests are focused on algorithmic differentiation, source analysis and source transformations facilitated by compiler frameworks with a focus on the C++ programming language.