FuzzyFlow: Leveraging Dataflow To Find and Squash Program Optimization Bugs

Philipp Schaad
philipp.schaad@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Timo Schneider
timos@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Tal Ben-Nun
talbn@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Alexandru Calotoiu
acalatoiu@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Alexandros Nikolaos Ziogas
alziogas@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Torsten Hoefler
htor@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

ABSTRACT

The current hardware landscape and application scale is driving performance engineers towards writing bespoke optimizations. Verifying such optimizations, and generating minimal failing cases, is important for robustness in the face of changing program conditions, such as inputs and sizes. However, isolation of minimal test-cases from existing applications and generating new configurations are often difficult due to side effects on the system state, mostly related to dataflow. This paper introduces FuzzyFlow: a fault localization and test case extraction framework designed to test program optimizations. We leverage dataflow program representations to capture a fully reproducible system state and area-of-effect for optimizations to enable fast checking for semantic equivalence. To reduce testing time, we design an algorithm for minimizing test inputs, trading off memory for recomputation. We demonstrate FuzzyFlow on example use cases in real-world applications where the approach provides up to 528 times faster optimization testing and debugging compared to traditional approaches.

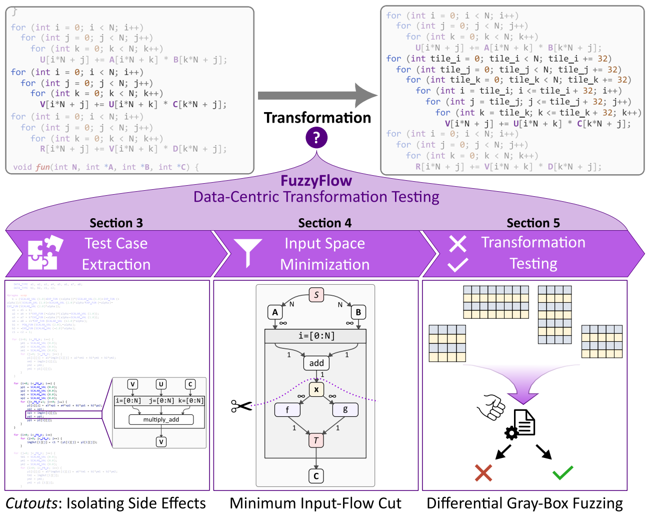
CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; **Software defect analysis**; *Compilers*; Development frameworks and environments.

KEYWORDS

software testing, fuzzing, translation verification, test generation

ACM Reference Format:
Philipp Schaad, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Alexandros Nikolaos Ziogas, and Torsten Hoefler. 2023. FuzzyFlow: Leveraging Dataflow To Find and Squash Program Optimization Bugs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3613214



```
}
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      U[i*N + j] += A[i*N + k] * B[k*N + j];
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      V[i*N + j] += U[i*N + k] * C[k*N + j];
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      R[i*N + j] += V[i*N + k] * D[k*N + j];

void fun(int N, int *A, int *B, int *C) {
```

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      U[i*N + j] += A[i*N + k] * B[k*N + j];
for (int tile_i = 0; tile_i < N; tile_i += 32)
  for (int tile_j = 0; tile_j < N; tile_j += 32)
    for (int tile_k = 0; tile_k < N; tile_k += 32)
      for (int i = tile_i; i <= tile_i + 32; i++)
        for (int j = tile_j; j <= tile_j + 32; j++)
          for (int k = tile_k; k <= tile_k + 32; k++)
            V[i*N + j] += U[i*N + k] * C[k*N + j];
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      R[i*N + j] += V[i*N + k] * D[k*N + j];
```

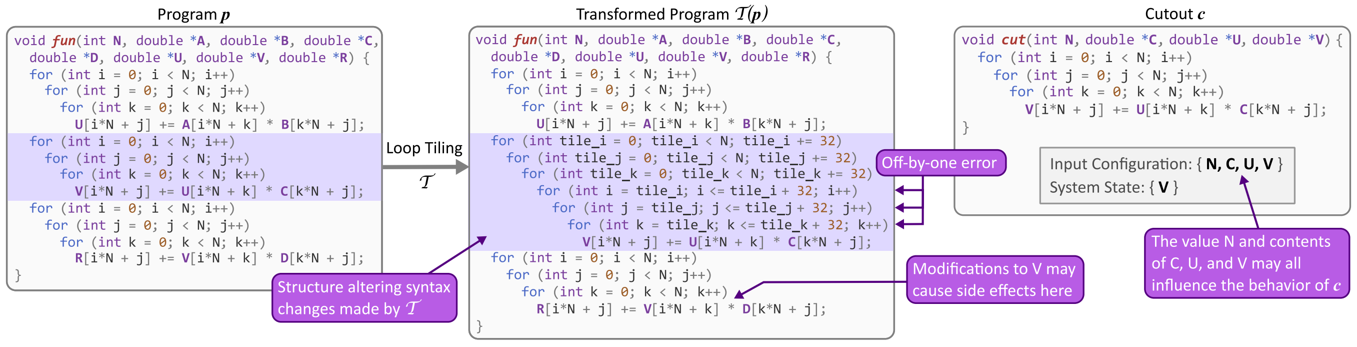
Figure 1: Overview of FuzzyFlow.

1 INTRODUCTION

Automatic compiler optimizations play an important role in getting good performance out of modern scientific applications. However, the limited knowledge a compiler has over certain program parameters can cause overly conservative optimizations. Applications are often further fine-tuned manually by performance engineers for specific input data or hardware architectures to exploit more potential performance gains. Manual application of such repetitive optimization patterns can be time consuming, which is why performance engineers may provide compilers with purpose-built, more aggressive optimization passes to be applied to a program at a large scale [8, 27, 31].

While optimizing compilers are well-studied and tested tools, with a lot of research going into making sure their code transformations preserve program semantics [29, 32, 33], the same does not hold true for custom optimization passes. To apply such a custom transformation indiscriminately, it consequently has to go through rigorous testing. Unfortunately, writing hundreds of test cases for a transformation is both labor intensive and may still miss crucial corner cases.

To avoid this issue, transformation rules themselves may be formally verified to ensure they preserve program semantics [33, 44,


Program p
Loop Tiling
$\mathcal{T}$
Structure altering syntax changes made by $\mathcal{T}$
Transformed Program $\mathcal{T}(p)$
Off-by-one error
Modifications to V may cause side effects here
Cutout c
The value N and contents of C, U, and V may all influence the behavior of c

```
void fun(int N, double *A, double *B, double *C,
  double *D, double *U, double *V, double *R) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        U[i*N + j] += A[i*N + k] * B[k*N + j];
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        V[i*N + j] += U[i*N + k] * C[k*N + j];
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        R[i*N + j] += V[i*N + k] * D[k*N + j];
}
```

```
void fun(int N, double *A, double *B, double *C,
  double *D, double *U, double *V, double *R) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        U[i*N + j] += A[i*N + k] * B[k*N + j];
  for (int tile_i = 0; tile_i < N; tile_i += 32)
    for (int tile_j = 0; tile_j < N; tile_j += 32)
      for (int tile_k = 0; tile_k < N; tile_k += 32)
        for (int i = tile_i; i <= tile_i + 32; i++)
          for (int j = tile_j; j <= tile_j + 32; j++)
            for (int k = tile_k; k <= tile_k + 32; k++)
              V[i*N + j] += U[i*N + k] * C[k*N + j];
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        R[i*N + j] += V[i*N + k] * D[k*N + j];
}
```

```
void cut(int N, double *C, double *U, double *V) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        V[i*N + j] += U[i*N + k] * C[k*N + j];
}
```

Input Configuration: { **N, C, U, V** }
System State: { **V** }

Figure 2: Off-by-one error in tiling a matrix-matrix multiplication in a matrix chain multiplication $R = ((A \cdot B) \cdot C) \cdot D$.

48, 52]. However, the correctness of certain transformations may depend on specific parameters or program inputs [8], making it difficult for this technique to detect issues related to changing program conditions. Checking for semantic preservation through symbolic execution and concolic testing [9, 22], or fuzzing and random testing [7, 36, 64] addresses this issue and allows transformations to be checked under various conditions instead. Unfortunately, this strategy is often impractical in the context of long-running scientific computing applications such as numerical weather prediction or machine learning, where a single program execution can consume thousands of node hours and problem sizes may be fixed.

Ideally a small test case that captures the changes made by a transformation can be extracted from the application, which allows equivalence checks to complete much faster. However, capturing all semantic changes or *side effects* such changes may introduce is often difficult due to program patterns such as pointer aliasing.

In this paper we specify the requirements for localized transformation validation and propose *FuzzyFlow*: a framework for testing transformations that leverages parametric dataflow program representations to extract minimal test cases capturing all transformation side effects, summarized in Fig. 1. By exploiting how dataflow languages expose the true data read-write set of each operation, it is possible to obtain the full system state of a test case for a given set of inputs and parameters. Using differential testing [39] we can then check that transformations do not alter this system state for the same set of input parameters. To uncover input-dependent faulty behavior we employ gray-box fuzzing [20] to sample different input configurations for testing. We devise a dataflow-graph analysis technique that minimizes the test case input space at the cost of recomputation to reduce this testing effort. With this approach we report any fault-inducing transformation instance and generate a fully reproducible, minimal test case including inputs that can aid in debugging transformations.

We demonstrate the effectiveness of this approach by providing a proof-of-concept implementation of FuzzyFlow that verifies custom transformations written in the DaCe optimization framework [4]. Using this implementation on a set of real-world scientific applications, we showcase how our technique enables fast optimization testing, and how the generated test cases enable simplified debugging on consumer workstations when optimizing HPC applications.

In summary, this paper makes the following contributions:

- Requirements specification for localized transformation verification on general programs.
- Data-centric method for extracting *minimal* test cases capturing all side effects of structural program changes to verify semantics preservation with differential testing.
- Graph-analytical approach to minimize test case inputs and reduce verification effort.
- A practical tool enabling up to 528 times faster optimization testing and debugging in real-world use cases.

2 LOCALIZED OPTIMIZATION VERIFICATION

Optimizations are a special form of program *transformations* that alter a program in an attempt to improve performance. More generally, a program transformation $\mathcal{T} : \mathcal{P} \to \mathcal{P}$ is a function that takes a program $p \in \mathcal{P}$ and applies a change to arrive at a different program $p' = \mathcal{T}(p)$. Changes made by a transformation may be contained to individual basic blocks, or they may alter the program structure. In addition to that, a general transformation may operate *within* program representations, such as with a source-to-source transformation, or it may transform a program from one representation into a different one.

Optimizations are a special subset of such transformations that typically operate within representations, often on a compiler's intermediate representation (IR), and aim to produce a program p' that is semantically equivalent to the starting program p. We say that two programs p and p' are semantically equivalent if, for a space of possible input data and parameters $\mathcal{S}_p$, it holds that $\forall s \in \mathcal{S}_p : p(s) = p'(s)$. We denote this as $p \mathrel{\widehat{=}} p'$, and semantic inequality as $p \mathrel{\widehat{\neq}} p'$.

The tiling optimization shown in Fig. 2 is an example of such a structure-altering source-to-source transformation. The function shown computes a matrix chain multiplication $R = ((A \cdot B) \cdot C) \cdot D$ with four $N \times N$ matrices. By tiling the loops that perform the second multiplication in the chain, the transformation aims to improve memory reuse but keep the product of the computation intact. However, the transformation introduces a bug. The loop conditions of the tiled loops are incorrectly using a <= comparison instead of <. This off-by-one error changes the semantics of the program. Executing the application would expose this problem, but if the multiplication is part of a larger application, that becomes costly.

Instead, the transformation can also be verified in the given example by only extracting the second matrix-matrix multiplication into a separate sub-program c, shown on the right in Fig. 2. By checking that for any two $N \times N$ matrices U and C, the results in V are the same before and after tiling we can catch the erroneous transformation in less time. This is possible because out of all the memory locations being written to inside of c, only the contents of V can have an effect on the semantics of the third multiplication. As such, we call V the output or **system state** of c. Similarly, the contents of the scalar value N, as well as the matrices C, U, and V may influence the program behavior *inside* of c and consequently may influence the results of the computation. We refer to this as the **input configuration** of c.

We call such a sub-program $c \subseteq p$ with a clear input configuration and system state a **cutout** of p. With a given input configuration, a cutout c can be treated as a stand-alone, executable program. Because any change to c that results in a change to the system state for the same input configuration indicates a change to the semantics of c, we can use cutouts to check for an optimizing transformation's preservation of semantics.

Specifically, for a given transformation $\mathcal{T}$ we want to find and extract a cutout c from p to capture all structural program changes made by $\mathcal{T}$ and then isolate possible side effects by capturing the input configuration and system state of c. Using differential testing on c and $\mathcal{T}(c)$ we can then check whether $c \mathrel{\widehat{=}} \mathcal{T}(c)$ holds by checking for changes to the system state. Since the system state includes everything that can affect the remainder of the program, it then follows that $c \mathrel{\widehat{=}} \mathcal{T}(c) \implies p \mathrel{\widehat{=}} \mathcal{T}(p)$ and $c \mathrel{\widehat{\neq}} \mathcal{T}(c) \implies p \mathrel{\widehat{\neq}} \mathcal{T}(p)$.

Since large-scale algorithmic restructurings to p may entail changes where the resulting cutout c is not much smaller than the original application, this approach is particularly well suited for more local optimizations, including loop-level compiler optimizations and peephole optimizations. Local optimizations such as the loop-level tiling optimization demonstrated in Fig. 2 apply smaller changes that can result in $c \lll p$, which leads to fast turnaround times during testing. However, ensuring $c \lll p$ comes with a set of challenges. Especially while still identifying any changes in program semantics and capturing all possible transformation side effects in the system state of c. To address these challenges, the program representations used for extracting cutouts should fulfill a set of requirements. We discuss these requirements and how different program representations fulfill them in the following subsections and provide an overview of them in Table 1.

2.1 Generalization

The first challenge with extracting and using cutouts to check for a transformation's correctness comes from the fact that some transformations may behave correctly for certain inputs, but not for others. For example, even if the off-by-one error in the tiling optimization from Fig. 2 is fixed, the transformation still contains a bug. Specifically, due to the way the inner, tiled loops are bounded, the optimized program will cause out of bounds memory accesses for any inputs where N is not a multiple of the tile size 32.

To detect this kind of input-dependent faulty behavior, a transformation's cutout should be evaluated under a series of different inputs and input sizes, or through symbolic execution. However, not all possible inputs are considered valid inputs. Certain values,

Table 1: Requirements for localized optimization testing.

Requirements ➜	Side Effect Analysis			Generalization	
Representation ➜	Scalar	Memory	Sub-region	Inputs	Sizes
Abstract Syntax Tree (AST)	✗	✗	✗	✗	✗
SSA-Form [30]	✓	✗	✗	✗	✗
PDG [19]	✓	✓	✗	✗	✗
MLIR [31]	✓	✓	✓†	✓	✗
Parametric Dataflow [4, 28]	✓	✓	✓	✓	✓

† Constant sizes only.

such as pointers, may be used to indirectly address other memory, and changes to them can cause invalid memory accessing or undefined behavior. Both in differential testing and symbolic execution, this can lead to a large number of false positives, making testing impractical. As such, it is important that for a given cutout, we can analyze what input values may be modified to what extent to generalize test cases for many different inputs.

While difficult in SSA form representations such as LLVM-IR [30], MLIR [31] dialects make it easy to detect when certain values are being used to reference other memory locations. However, generalizing arbitrary cutouts to varying input *sizes* is challenging in most program representations due to missing context. For example, a pointer to a variable sized array tells us nothing about the size of the array without additional information. This is demonstrated in the extracted cutout for our tiling optimization on the right side of Fig. 2. By looking at the input configuration, the dependency between the scalar value N and the size of the memory region pointed to by the pointer C is opaque. Both N and the pointer are seemingly independent inputs, and evaluating it under different input sizes or symbolic execution would primarily cause segmentation faults or undefined behavior, causing false positives.

To avoid this, a representation used to extract cutouts should be *parametric*. This means that data containers are not expressed as pointers with unknown size, but instead with a known size and shape given by an expression e, where e can be constant or dependent on certain program parameters. In our tiling example that means the size of C would be given with the expression $N \times N$, which re-establishes the relationship between the parameter N and the data size, enabling testing with different sizes.

2.2 Side Effect Analysis

A further challenge arises when trying to determine the input configuration and system state for a program cutout based on the structural changes made by a transformation. Since structural changes inside the cutout may affect semantics in the remainder of the application, it is important that the cutout's system state captures such changes. In the tiling example from Fig. 2, the incorrect optimization causes different values to be computed for V than before tiling, in turn affecting the results in R. The change to the computation of R is thus considered a *side effect* of our transformation and the change to how V is computed. V must therefore be part of the cutout's system state.

Correctly determining the cutout input configuration and system state consequently requires detecting any such side effects. This form of side effect analysis is more challenging in some program representations than in others. Even the change of a single variable

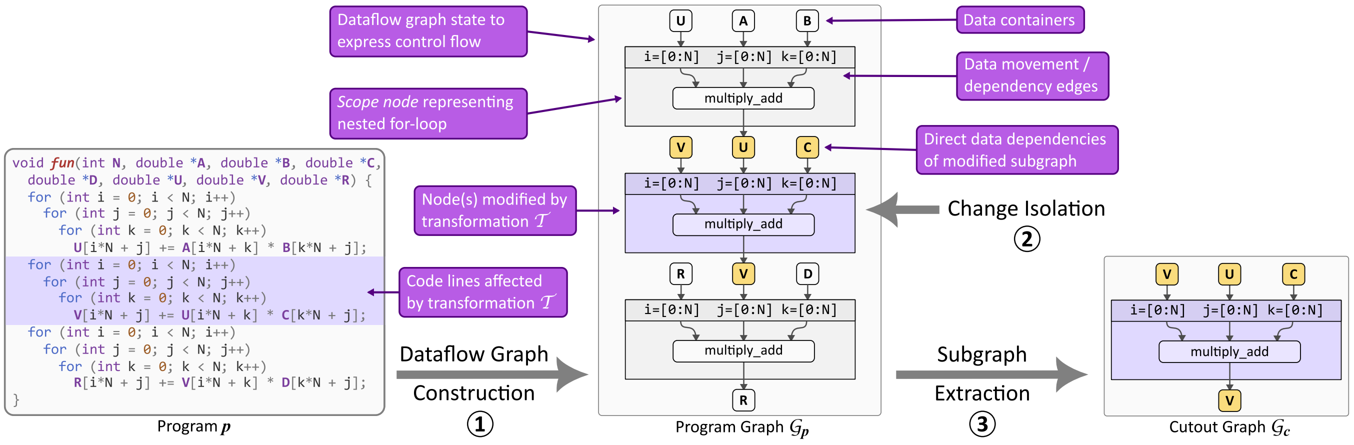

```
void fun(int N, double *A, double *B, double *C,
  double *D, double *U, double *V, double *R) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        U[i*N + j] += A[i*N + k] * B[k*N + j];
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        V[i*N + j] += U[i*N + k] * C[k*N + j];
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
        R[i*N + j] += V[i*N + k] * D[k*N + j];
}
```
Program p
Dataflow graph state to express control flow
Scope node representing nested for-loop
Node(s) modified by transformation $\mathcal{T}$
Code lines affected by transformation $\mathcal{T}$
Dataflow Graph Construction ①
Data containers
Data movement / dependency edges
Direct data dependencies of modified subgraph
Change Isolation ②
Subgraph Extraction ③
U A B V U C R V D
i=[0:N] j=[0:N] k=[0:N]
multiply_add
Program Graph $\mathcal{G}_p$
Cutout Graph $\mathcal{G}_c$

Figure 3: Cutout extraction procedure for an arbitrary local optimization, demonstrated with a loop tiling transformation.

value introduced through a program change may have side effects outside of the cutout and thus require said variable to be part of the system state. The side effects arising from changes to a single register value or *scalar* are easily exposed in static single assignment (SSA) form program representations, such as LLVM IR [30], as shown in Table 1.

However, it is harder to catch possible *memory* side effects. Program patterns such as pointer aliasing can mask the true effects of program statements. Catching such aliasing or masking effects in source code, AST representations, or even SSA form representations is difficult without advanced and costly pointer analysis. Representations like program dependence graphs (PDGs) [19] and other dataflow IRs expose the data dependencies between individual statements, simplifying this side effect analysis drastically.

Pointers also make it challenging to detect side effects related to accesses in specific memory *sub-regions*. This includes changes that affect overlapping memory regions or indexing changes which cause out of bounds accesses. Given the difficulties of pointer analysis, these side effects are hard to detect in AST or SSA form program representations. Even with the help of a PDG it is difficult to detect side effects related to overlapping memory regions. Some built-in dialects in MLIR [31] enable this form of sub-region or index analysis, as long as structs and arrays have constant sizes known during compilation. Since parametric program representations keep the relationship between data containers and their size intact, they enable such analyses even for symbolic struct or array sizes where the concrete sizes are not statically known.

2.3 Parametric Dataflow

The memory side effect analysis capabilities of dataflow IRs such as PDGs exist together with the generalization and sub-region analysis capabilities of parametric representations in *parametric dataflow representations*. This gives representations such as HPVM [28], Naiad [42], stateful dataflow multigraphs (SDFGs) [4], or Dryad [25] all the properties necessary to extract generalizeable program cutouts that isolate all side effects of program transformations.

We construct a reference implementation of our approach for the SDFG representation that is used by the optimization framework DaCe [4]. This framework has become a popular choice for optimizing scientific high performance computing applications from numerous fields, where engineers write purpose-built, custom optimizing transformations to get high performance gains out of their applications [1, 5, 26, 70, 72]. Together with the ability to express arbitrary programs from Python, C, or Fortran, this makes the SDFG IR a good choice to demonstrate the effectiveness of our proposed approach. However, the techniques outlined in this paper are generally applicable to any parametric program representation adhering to the requirements outlined in Table 1.

Stateful Dataflow Multigraphs. Like most dataflow program representations, SDFGs are a graph-based program IR where graph nodes represent data containers and computations, and the edges connecting them represent data movement or dependencies. Each data movement edge is annotated with the exact data subset being accessed in the connected computations. Additionally, the graph is a *hierarchical* dataflow graph, with different hierarchies nested inside one another. Firstly, dataflow graphs are nested inside of states which are used to build a state machine that can express complex control flow. Secondly, constructs like for-loops are expressed with special *scope* nodes, where their loop body forms a nested dataflow graph inside of them. An example of this IR can be seen in Fig. 3, which provides an overview of the cutout extraction procedure.

3 TEST CASE EXTRACTION

Given a transformation $\mathcal{T}$ and program p, extracting a program cutout c with the help of a graph-based IR that exposes dataflow can be thought of as extracting the dataflow subgraph which is modified by the transformation. We split this procedure into three distinct steps summarized in Fig. 3:

① *Dataflow Graph Construction.* The first goal is to translate the program p into its dataflow graph representation $\mathcal{G}_p$. We can leverage DaCe [4] for this purpose, which can construct parametric dataflow graphs from a variety of high level languages.

② *Change Isolation.* Next, the changes made by a transformation need to be identified to get a starting point for what to extract into a cutout. In a graph-based IR, this corresponds to identifying all graph nodes that have changed between the two program graphs $\mathcal{G}_p$ and $\mathcal{G}_{\mathcal{T}(p)}$. If the change includes modified, added, or removed edges, both the edge source and destination nodes are considered to be modified. We denote this set of modified nodes as $\Delta_{\mathcal{T}}$. When treating transformations as black boxes, this change set has to be obtained through analyzing the difference between $\mathcal{G}_p$ and $\mathcal{G}_{\mathcal{T}(p)}$.

However, in some optimization frameworks, such as DaCe, transformations readily expose the set of changes made by reporting what specific program patterns they modify or by specifying rewrite rules. Such white-box transformations thus do not require a difference analysis between $\mathcal{G}_p$ and $\mathcal{G}_{\mathcal{T}(p)}$. Since the cutout c where the transformation will be tested acts as an entirely separate program isolated from p, all effective changes made by a transformation are tested, even if the transformation incorrectly self-reports the change set. If a transformation attempts to perform a change not captured by c, this will lead to a crash or exception since c does not contain the corresponding element, thus exposing a problem.

③ *Subgraph Extraction.* After identifying all affected dataflow graph nodes $\Delta_{\mathcal{T}}$, the corresponding subgraph can be extracted into a separate program. We first construct a new, empty program graph $\mathcal{G}_c = \emptyset$ for our cutout c. Now we can create a copy for each node $n \in \Delta_{\mathcal{T}}$ and add it to $\mathcal{G}_c$. For every node n, we look at each incoming and outgoing edge connected to it in the dataflow graph. If n performs a computation, these connected edges in the dataflow graph expose what data containers are being accessed and at what indices or sub-regions. If any of these data containers are not part of $\mathcal{G}_c$ yet, we copy them over as well. This ensures that all direct data dependencies for the nodes affected by $\mathcal{T}$ are part of $\mathcal{G}_c$, including all properties of the corresponding data containers, such as their size.

When copying over data dependencies, we can additionally leverage the parametric properties of the dataflow representation to minimize the size of data containers required in $\mathcal{G}_c$. Since parametric dataflow representations such as SDFGs allow us to analyze exactly what data indices or sub-regions are being accessed by each operation, we can avoid having to include entire data containers if only sub-regions are needed to capture dependencies. For example, if a computation modified by $\mathcal{T}$ only accesses the indices 0 to 9 of an array `my_arr` of size N, only the first 10 elements of `my_arr` need to be included in $\mathcal{G}_c$. This helps reduce both the size and memory consumption of our cutout c, consequently speeding up the testing process.

Fig. 3 shows the cutout dataflow graph $\mathcal{G}_c$ that we can obtain by following this procedure based on the incorrect tiling transformation introduced in Section 2 (see Fig. 2).

3.1 Capturing Side Effects

Now that the dataflow graph $\mathcal{G}_c$ for our cutout c contains everything directly affected by the $\mathcal{T}$, including all direct data dependencies, we need to make sure all possible side effects of $\mathcal{T}$ are captured. This means we must correctly determine the **system state** the cutout c leaves after its execution. The system state consists of any data container or subset thereof written to inside the cutout, that may be read again elsewhere after the cutout's execution, if c were placed back inside the original program p. Any data container where this is the case can have an influence on the behavior of p and consequently its results may affect semantics.

There are two ways in which data may be read again after c was executed in the context of p, and that consequently need to be checked with two analyses:

External Data Analysis. Any write operations c performs to external or persistent data, such as program outputs or files on disk, always require the corresponding data container to be part of the system state. In SDFGs, data containers helpfully have a property that indicates whether their allocation lifetime is *transient* or not. Any data container that is not marked as transient is not managed by the program and may persist, consequently leaving the chance to be read after the program has exited.

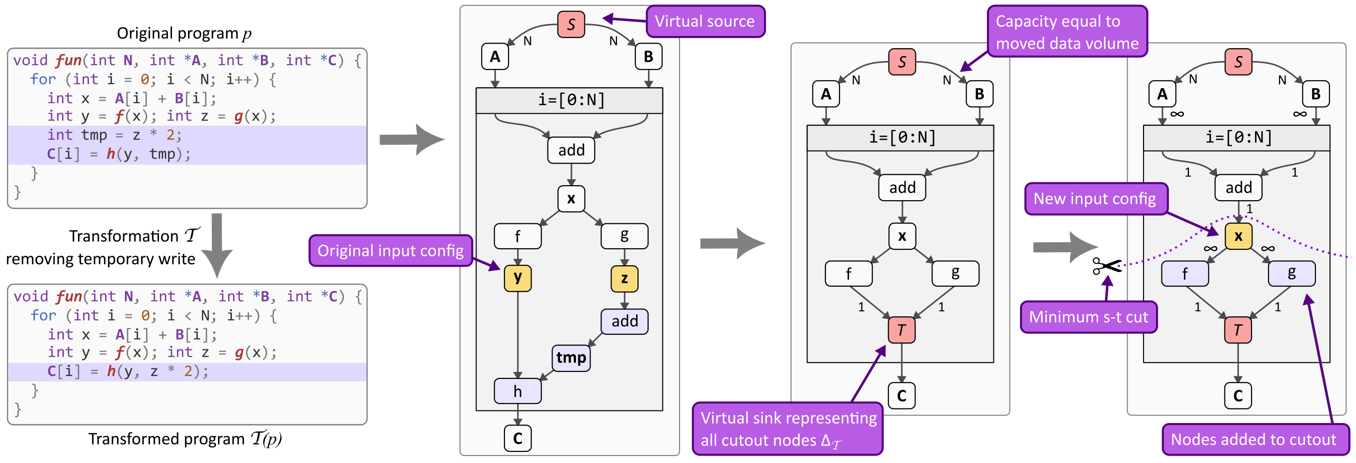
Program Flow Analysis. Additionally, a write operation to a data container d inside of c requires d to be part of the system state, if there is a read operation from d inside of $\bar{c} = p - c$ which is reachable from c. To check for these side effects, we first analyze each write operation performed inside of $\mathcal{G}_c$ together with the sub-range or indices written. We then perform a breadth first search (BFS) through the original program graph $\mathcal{G}_p$, starting at the nodes $n \in \Delta_{\mathcal{T}}$ that created the cutout. For any read operation we encounter, we check if the data container being read is part of the set of write operations performed inside of $\mathcal{G}_c$, and if the read range or indices overlap with any of the written ranges or indices. If yes, the data container is added to the system state of c.

After these dataflow analyses, anything that is not marked as part of the system state does not have any influence on the semantics of p and consequently does not need to be compared during the testing procedure. There are some possible side effects that are not easily exposed even through dataflow representations, such as through the use of user-defined callbacks or libraries that may produce side effects. Fortunately, their use can easily be detected, enabling clear warnings when transformations can produce unintended side effects that may not be caught.

3.2 Determining Input Configurations

To properly test a transformation's cutout under multiple circumstances, we must also extract everything that may be part of the **input configuration** for our cutout c. The input configuration consists of all data containers that may already contain data before c is executed, and may consequently influence the behavior of c. We can obtain this input configuration with two analyses analogous to the ones performed for the system state:

In the external data analysis, we add the data containers of any read operations from external or persistent data inside of c (data containers not marked as transient) to the input configuration. In the subsequent program flow analysis, we analyze each read operation performed inside of $\mathcal{G}_c$ and take note of the read sub-ranges and indices. By performing a reversed BFS starting at the nodes $n \in \Delta_{\mathcal{T}}$, we add any write operation to the input configuration if there is a corresponding read to an overlapping index inside of $\mathcal{G}_c$. This uncovers any write operations that may reach the cutout through some execution path and consequently ensures that anything that may already contain data when the cutout is executed, is part of the input configuration.


Original program p
Transformation $\mathcal{T}$ removing temporary write
Transformed program $\mathcal{T}(p)$
Virtual source
Original input config
Capacity equal to moved data volume
Virtual sink representing all cutout nodes $\Delta_{\mathcal{T}}$
New input config
Minimum s-t cut
Nodes added to cutout
S
A
B
N
i=[0:N]
add
x
f
g
y
z
tmp
h
C
T
1
∞

```
void fun(int N, int *A, int *B, int *C) {
  for (int i = 0; i < N; i++) {
    int x = A[i] + B[i];
    int y = f(x); int z = g(x);
    int tmp = z * 2;
    C[i] = h(y, tmp);
  }
}
```

```
void fun(int N, int *A, int *B, int *C) {
  for (int i = 0; i < N; i++) {
    int x = A[i] + B[i];
    int y = f(x); int z = g(x);
    C[i] = h(y, z * 2);
  }
}
```

Figure 4: Overview of the steps performed during the minimum input-flow cut process to reduce a cutout's input configuration.

4 MINIMIZING INPUT CONFIGURATIONS

For realistic cutouts, the size of the input configuration space can be very large. Complex transformations in large applications can result in cutouts where the input configuration may consist of dozens of data containers. During the differential testing to check whether a transformation is correct, we want to cover as much of this input configuration space as possible to make it unlikely that a fault inducing input goes undiscovered. It is thus desireable to keep the size of this input configuration space as small as possible to reduce the testing effort required.

Fortunately, there are many situations where the input configuration can be reduced significantly from its initial size. Consider for example the program shown on the left of Fig. 4. A transformation attempts to subsume the computation `z * 2` into the function call to `h` to get rid of the write to the temporary data container `tmp`. The cutout constructed through the procedure outlined before will include the multiplication and the function call to `h`. The input configuration would consequently consist of both the data containers `y` and `z`. However, by also including the function calls to `f` and `g` in the cutout, the input configuration only needs to contain the data container `x`. This halves the input space and with it the effort required when testing different input configurations, at the cost of some additional computation.

4.1 A Graph-Analytical Solution

To automatically minimize the input configuration space, we devise a method to gradually expand the cutout with surrounding dataflow, until we find a cutout with a smaller input configuration space. If the input space cannot be further minimized, the original cutout is used. Since the testing effort only depends on the size of the input configuration space, we can refrain from minimizing the system state of the cutout. Consequently, we can ignore anything that happens in the program after the cutout, and only need to consider including more of the cutout's predecessor nodes in the graph.

Since the edges in a dataflow graph represent data movement, they have a certain data *volume* associated with them, which is being moved or accessed across them. For all data containers that are part of our input configuration, that data movement volume used to access them thus corresponds to the amount of data needed to sample a single input configuration. We can think of this data movement volume as the *capacity* of each data movement edge.

This can be exploited to reformulate finding the optimal solution for minimizing the input configuration space as finding the minimum s-t cut in the dataflow graph between the start of the program and our cutout. In a flow graph, the minimum s-t cut refers to a partitioning of the graph between a source node and a sink node into two disjoint components, such that the sum of edge capacities crossing between the two components is minimal. In our context, a cut satisfying that property translates to a cut through the dataflow graph that minimizes the volume of data being moved between components. This minimal flow of input data into the cutout component thus minimizes the size of the input configuration space.

4.2 Minimum Input-Flow Cut

The procedure for finding a cut between the start of the program in $\mathcal{G}_p$ and our cutout graph $\mathcal{G}_c$ that minimizes this *input-flow* can be split into two separate phases:

Preparation. Since the dataflow graph $\mathcal{G}_p$ may have more than one source node, i.e., nodes without incoming edges, there is no unique start node that can be used for the minimum cut. To address this, we insert a dummy source node S into $\mathcal{G}_p$ and connect it to each node which does not have any incoming edges. If the node in question is a data node, we give the edge a capacity equal to the size of the data container that node represents. Similarly, the cutout may consist of more than one graph node, meaning that there is no unique sink in $\mathcal{G}_p$ to represent the cutout either. Instead, we insert a dummy sink node T into $\mathcal{G}_p$ to represent $\mathcal{G}_c$ as one node.

Next, we create edges between S and each data node that accesses external or persistent data. The capacity of those edges is set to the size of the data containers being accessed by each node. All other incoming edges to those data nodes have their capacity set to ∞. This is necessary since each access to external or persistent data is always part of the input configuration, as discussed in Sec. 3.2.

We can now look at each data node inside of $\mathcal{G}_c$ that belongs to the initial input configuration of the cutout. For each such node, we redirect all incoming edges such that their destination is set to the sink node T instead. Additionally, the capacity of these edges is set to the volume moved across them, which we can derive by looking at the data sub-range or indices being accessed through them.

To complete the process of connecting our virtual sink node T, we now look at the outgoing edges for each node in $\mathcal{G}_c$. For any such edge where the destination lies outside of $\mathcal{G}_c$, we have two situations to consider: either the destination has a path back to a node which lies inside of $\mathcal{G}_c$, or it does not. If there is no path back to $\mathcal{G}_c$, then we simply change its source to T. However, if there is a bath back to $\mathcal{G}_c$, we can redirect the edge to lead from S to T with a capacity of 0. This is possible because any data the edge accesses will already be part of the cutout, and consequently does not need to be added to the input configuration. The data movement volume across it is consequently considered 'free' in the context of our minimum input-flow cut. After making these connections, we remove all nodes in $\mathcal{G}_c$ and any connected edges from $\mathcal{G}_p$.

Finally, we run over all data nodes in $\mathcal{G}_p$ and set the capacity of all of their outgoing edges to ∞. We do this since a cut over one of those edges would sever a data dependency without adding the corresponding data node to the cutout component. Instead, we want a potential cut to happen before the data node. This preparation process is demonstrated on the transformation removing temporary writes shown in Fig. 4.

Minimum S-T Cut. In the second phase, we use the program graph $\mathcal{G}_p$ prepared in the previous phase to solve the minimum s-t cut problem and find a cutout with a reduced input configuration space. The max-flow min-cut theorem [13] states that the maximum flow through a flow graph (or network) is given by the minimum s-t cut through the graph. This means that we can use an existing efficient implementation of the Ford-Fulkerson algorithm [21] for finding the maximum flow from S to T, such as Edmonds-Karp [15], to determine our minimum cut.

Up until now, the capacity of each edge may still be given as a symbolic expression due to the parametric nature of the dataflow representation. Since such maximum flow algorithms don't work with symbolic capacities, we concretize the symbol values at this point in the process with constant values that may be provided by the user.

For $\mathcal{G}_p$ with the set of nodes V and the set of edges E, the Edmonds-Karp maximum flow algorithm finds the minimum cut with a time complexity of $O(|E|^2|V|)$. Given the minimum s-t cut, we can now extend the cutout by everything that resides in the same component of T and can reach T, to arrive at a cutout with the smallest possible input configuration space.

5 DIFFERENTIAL TESTING

With an obtained minimal cutout c that captures the changes and side effects of a transformation $\mathcal{T}$, testing whether $\mathcal{T}$ preserves semantics can now be done by checking that $c \cong \mathcal{T}(c)$ holds. If we recall from Sec. 2, for a cutout c with a space of possible input data and parameters $\mathcal{S}_c$, $c \cong \mathcal{T}(c)$ can be shown by showing:

$$\forall s \in \mathcal{S}_c : c(s) = c'(s),\ \text{where } c' = \mathcal{T}(c)$$

However, even after minimizing the size of the input configuration space, the number of possible input configurations $|\mathcal{S}_c|$ may still be high. Even for a trivially small cutout that adds two 32-bit integers together, $|\mathcal{S}|$ is equal to 2^{64}. Verifying each of these input configurations would take an impractically long time or may be entirely infeasible.

Symbolic Execution. A popular approach to avoid having to run all possible input configurations is to employ symbolic execution or concolic testing [9, 22]. Symbolic execution is a widely used technique that has found a strong foothold in software and security testing. However, symbolic execution faces a few challenges that make it impractical for the context of scientific high performance computing applications.

Specifically, symbolic execution struggles with anything where it does not have direct source knowledge, such as library or system calls, and may suffer from state space explosion in the context of nested loops [2]. Unfortunately, both of these challenging situations are encountered regularly in high performance, scientific code. Substitution of computations with architecture-specific intrinsics or nested loop optimizations such as tiling are often the subject of program optimizations in supercomputing contexts.

In addition to that, most scientific applications heavily rely on floating point arithmetic in their computations. While there are techniques to perform symbolic execution in the context of floating point operations, they currently still lack the robustness to efficiently and reliably catch issues. Most widely used frameworks such as KLEE [9] thus do not support floating point symbols yet. These limitations make symbolic execution a less desirable technique in the context of scientific HPC applications.

5.1 Differential Fuzzing

Instead of using symbolic execution, we can use fuzzing [36] to randomly sample input configurations from $\mathcal{S}_c$ and evaluating them over a series of $t \lll |\mathcal{S}_c|$ trials. This still avoids having to test each individual input configuration, and instead samples inputs in such a way that the likelihood of finding issues is maximized.

In each trial, an input configuration $s \in \mathcal{S}_c$ is run through both the original cutout c and $\mathcal{T}(c)$. For each such trial, the system state of the two cutouts after execution is compared and any system state variation $c(s) \neq c'(s)$ labels the transformation as invalid. Specifically, a change in the system state is reported if either the transformed program c' crashes or hangs while c does not, or the numerical results produced differ by more than a given threshold t_Δ. The threshold t_Δ can be configured by the user[1], and if t_Δ is not provided or set to 0, we instead test for bit-wise equality.

Gray-Box Fuzzing. To achieve a high probability of detecting a possible change in semantics, it is imperative that the configurations sampled over t trials cover the input configuration space well. This can be achieved by sampling uniformly at random from the space of all possible values for each input parameter, and doing so for all possible data types where they are not explicitly stated. This form of uniform sampling fully covers the input space, but comes with two additional challenges. Firstly, uniform sampling may lead to many uninteresting crashes, where both c and $\mathcal{T}(c)$ do not execute successfully. Secondly, the program may take different program paths based on specific input values, in which case

[1]For this paper we use $1e^{-5}$.

evaluating all possible program paths becomes statistically unlikely through uniform sampling.

To address the first challenge, we instead perform a form of gray-box fuzzing, meaning that we perform some amount of static analysis on both p and c to derive constraints for sampling. By deriving constraints for certain input parameters based on the context in which they are used we reduce the number of uninteresting crashes while simultaneously further reducing the space of possible input configurations. This in turn increases the likelihood with which a possible change in semantics can be uncovered.

We can perform two kinds of constraint analyses. First, we check if any input parameters are used to access other data containers inside of our cutout's dataflow graph $\mathcal{G}_c$. This is again where the parametric nature of the representation can help us by letting us analyze the sub-regions and indices accessed by each data movement edge. Any parameter that is used to access inside a data container is bound to the interval $[0, Dim_{max}]$, where Dim_{max} is the size of the data container in that specific dimension. For example, for an $N \times M$ matrix A, the access $A[i, j]$ would constrain i to the interval $[0, N]$ and j to $[0, M]$. A second analysis is performed on the original program's dataflow graph $\mathcal{G}_p$ to determine if any program context constrains parameters to a specific range. Of particular interest here are loop iteration variables that may be constrained to certain loop bounds, if the cutout was taken from inside one or more loops.

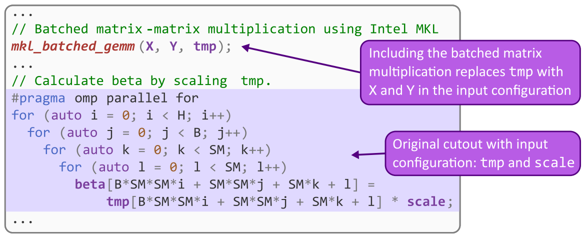
In addition to these derived constraints, an engineer may further constrain the testing process by providing custom constraints. This can be useful if it is known through domain knowledge that certain program parameters are always observed to have certain ranges. Finally, since a data container can never have a size of ≤ 0, any parameter that is used to determine the size of a data container is only sampled in the range $[1, Size_{max}]$, where $Size_{max}$ can be configured arbitrarily.

Coverage-Guided Fuzzing. To address the second challenge with uniform sampling, we can additionally use the obtained constraints to perform coverage-guided fuzzing. Here, the code is instrumented and program path coverage is recorded for each fuzzing trial. The fuzzer then attempts to mutate the sampled inputs in such a way that new program paths are taken in an attempt to maximize program coverage, consequently increasing the likelihood of finding bugs.

To enable this, an SDFG-based cutout c and its transformed counterpart $\mathcal{T}(c)$ can be turned back into C++ programs together with a small auto-generated harness that calls both programs and compares their outputs. If the cutout outputs differ, the harness causes a segmentation fault. This allows the use of existing coverage-guided fuzzers such as AFL++ [20] or other advanced fuzzing tools. While the powerful constraint analyses discussed before may not be performed this way, this allows us to profit both from the localized testing capabilities gained through cutouts, as well as the years of research into advanced fuzzing techniques [36].

6 CASE STUDIES

We demonstrate the use of FuzzyFlow in a real-world setting with case studies from working with and optimizing scientific HPC applications. To illustrate how the proposed technique helps test and debug transformations in HPC contexts on consumer workstations,



```
...
// Batched matrix -matrix multiplication using Intel MKL
mkl_batched_gemm(X, Y, tmp);
...
// Calculate beta by scaling  tmp.
#pragma omp parallel for
for (auto i = 0; i < H; i++)
  for (auto j = 0; j < B; j++)
    for (auto k = 0; k < SM; k++)
      for (auto l = 0; l < SM; l++)
        beta[B*SM*SM*i + SM*SM*j + SM*k + l] =
            tmp[B*SM*SM*i + SM*SM*j + SM*k + l] * scale;
...
```

Including the batched matrix multiplication replaces tmp with X and Y in the input configuration
Original cutout with input configuration: tmp and scale

Figure 5: Input space reduction for a loop nest cutout.

we run each test on a consumer-grade system with a 10-core Intel i9-7900X CPU at 4.5 GHz and 32 GB of RAM. We use Python 3.9, a development build of DaCe 0.14.2, GCC 10.3 and AFL++ 4.06a.

6.1 Minimizing Input Configurations

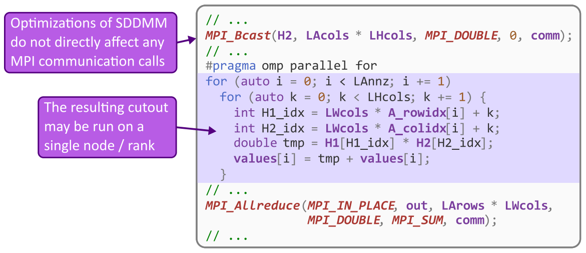
Machine learning workloads are an increasingly important part of the HPC landscape, consuming large amounts of compute resources for applications spanning diverse fields. With this case study we demonstrate how our minimum input-flow cut algorithm can significantly reduce the input space for extracted test cases while optimizing a machine learning application. For this purpose we select the encoder layer from the natural language model BERT [14]. This type of Transformer [57] is a widely used neural network, where even pre-trained models take hours to tune in large-scale distributed environments.

The Multi-Head Attention (MHA) in BERT's encoder layer contains a series of nested loops which perform a variety of element-wise operations, tensor contractions, and normalizations. Using a built-in transformation in DaCe, we can optimize most of these nested loops by vectorizing them. The transformation tiles the loops by the chosen vector size (4 by default) and augments the computations with vector instructions. Using FuzzyFlow, we test each instance of the transformation before applying to ensure program semantics remain unchanged. We optimize the application for use with the BERT$_{\text{LARGE}}$ [14] model size (batch size $B = 8$, attention heads $H = 16$, hidden size $N = 1024$, input/output sequence length $SM = 512$, embedding size $emb = 4096$, and projection size $P = \frac{I}{H} = 64$), which takes 12.1 seconds to run when accelerating BLAS operations with Intel MKL [24].

One instance of such a loop nest which performs the scaling of a tensor `tmp` is shown in Fig. 5. The resulting cutout capturing all structural changes from vectorizing the loop nest is highlighted in the figure and contains only the loop nest itself. The input configuration of this cutout initially consists of the $B \times H \times SM \times SM$ tensor `tmp` and the scalar value `scale`.

Using the minimum input-flow cut algorithm to reduce the size of the input space, we arrive at a larger cutout, which also includes the batched matrix-matrix multiplication that computes `tmp`. By including the computation of `tmp` it is no longer required as an input, and is instead replaced by the two $P \times B \times H \times SM$ tensors `A` and `B`. With the chosen program parameters this **reduces the input configuration by 75%**, consequently increasing test coverage.

The reduction in the input space additionally causes a 2× speedup in sampling input values and checking system state equivalence. Using AFL++ to perform coverage-guided fuzzing, this allows running an average of 43.7 fuzzing trials per second, which is **528 times**


Optimizations of SDDMM do not directly affect any MPI communication calls
The resulting cutout may be run on a single node / rank

```
// ...
MPI_Bcast(H2, LAcols * LHcols, MPI_DOUBLE, 0, comm);
// ...
#pragma omp parallel for
for (auto i = 0; i < LAnnz; i += 1)
  for (auto k = 0; k < LHcols; k += 1) {
    int H1_idx = LWcols * A_rowidx[i] + k;
    int H2_idx = LWcols * A_colidx[i] + k;
    double tmp = H1[H1_idx] * H2[H2_idx];
    values[i] = tmp + values[i];
  }
// ...
MPI_Allreduce(MPI_IN_PLACE, out, LArows * LWcols,
              MPI_DOUBLE, MPI_SUM, comm);
// ...
```

Figure 6: Cutout extraction enables testing on a single node.

faster compared to testing the transformation by running the entire application. When testing for different model sizes, AFL++ takes an average of 157 trials per vectorization optimization to discover that the transformation's correctness depends on specific input data sizes. By using our own gray-box fuzzing with parameter constraint analyses it only takes an average of 1 trial to uncover this input-dependent correctness, but at the cost of longer trial runs. The longer trial runs are primarily due to architectural setup and teardown penalties such as copying data between processes, but this overhead can be optimized by adopting strategies found in existing state of the art fuzzers such as AFL++.

6.2 From Multi-Node to Single-Node

Another category of increasingly important machine learning workloads are graph neural networks (GNNs) [6, 11, 62] such as Vanilla Attention [58], which lends itself well for running in a distributed setting. Vanilla Attention performs a Sampled Dense-Dense Matrix Multiplication (SDDMM) in each layer during forward propagation, which is the target of many optimization efforts due to its poor data locality [45, 65]. However, testing optimizations performed on SDDMM for correctness or debugging them can be challenging, especially in a distributed scenario. Tests that span multiple compute nodes are expensive to run, and not all applications can trivially be reduced to single-node contexts since behavior may vary between ranks or different communicator sizes.

By using the cutout-based testing process in FuzzyFlow we can effectively reduce test cases to a single rank for any optimizations that do not directly affect communication operations. An example of this is shown in Fig. 6, where extracting a cutout for SDDMM in Vanilla Attention allows testing on a single node. Since cutouts only include the direct data dependencies of the optimizing changes and detect side effects through changes in the cutout's system state, communication does not need to be included in a cutout if it is not directly modified. Any data received through collectives is subsequently exposed as regular data parameters to the cutout and subjected to differential fuzzing.

6.3 NPBench

Due to the increasing use of DaCe for program optimizations we test each of its built-in optimizations[2] on a set of micro-benchmarks. We use the benchmark suite NPBench [71], which contains a total of 52 benchmark programs from various application domains. For each benchmark application, we test each individual instance where any of the built-in optimizations can be applied. Most of the resulting *3,280 transformation instances* pass this testing procedure. However, we identify a total of **6 transformations** containing bugs and one transformation where correctness depends on specific combinations of input parameters and options passed to the transformations. We summarize our findings in Table 2.

[2]Some transformation requiring special hardware, such as FPGAs, are omitted.

Table 2: DaCe transformation bugs uncovered by FuzzyFlow.

Transformation	Failure
`BufferTiling`: Tiles buffers between loops	✗
`TaskletFusion`: Removes temporary writes	✗
`Vectorization`: Vectorizes loops	⚠
`MapExpansion`: Removes collapsing from parallel nested loops	🔨
`MapReduceFusion`: Removes intermediate buffers for reductions	🔨
`StateAssignElimination`: Program simplification	🔨
`SymbolAliasPromotion`: Program simplification	🔨

✗ Change in semantics ⚠ Input dependent 🔨 Generates invalid code

6.4 Optimizing Weather Forecasts

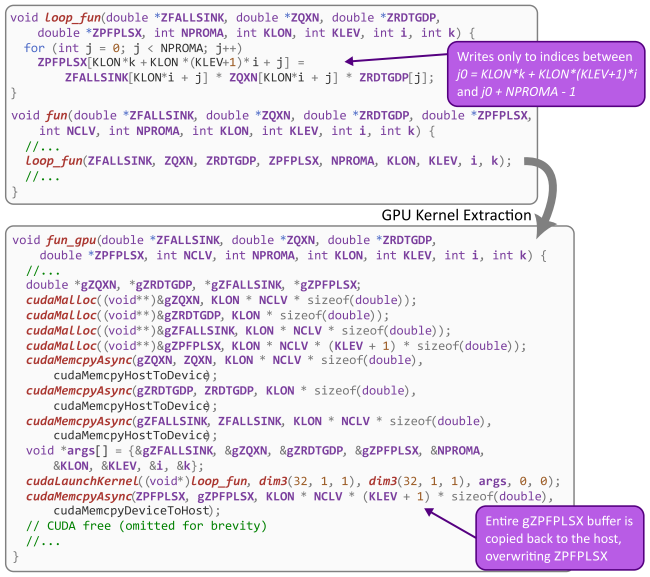
The cloud microphysics scheme (CLOUDSC) is part of the European Centre for Medium-Range Weather Forecasts' (ECMWF) Integrated Forecasting System (IFS) [17], and its Fortran implementation spans 3,163 lines of code. We worked with a group of engineers tasked with optimizing this application using DaCe by primarily exploiting more parallelism and porting the application to accelerators. In this process, the engineers wrote a series of transformations which were then applied to the application automatically.

While these transformations improved performance significantly, they also caused numerical errors in the computations. Due to the size of the application and the large number of transformations applied, identifying which transformation instances caused errors and then building minimal test cases to debug them was a time-consuming process. Just identifying a *single* transformation bug that incorrectly extracted a GPU kernel took one engineer *over 16 hours*. This was only possible by reducing the problem size, which often is not an option.

With this case study we demonstrate how the use of FuzzyFlow could help in such situations. Using FuzzyFlow we test each instance where the transformations used can be applied to the application over 100 trials. We do this to isolate which transformations contain bugs and find fault-exposing test cases to assist in debugging. We identify 3 transformations that cause semantic changes when applied to CLOUDSC at various stages of the optimization process.

Extract GPU Kernels. We first test a custom transformation written by the engineers while optimizing CLOUDSC, which automatically extracts GPU kernels from the application. It does this by identifying routines that may be run as GPU kernels and then generating CUDA code for those routines and inserting the necessary boilerplate such as kernel launch calls and data copies between the host and the GPU. We test a total of 62 instances of this transformation on CLOUDSC, out of which we identify **48 instances** that alter program semantics.

One of the failing test cases generated is shown in Fig. 7. A closer look reveals that the problem lies in the fact that the transformation generates data copies for the entire data containers touched by extracted GPU kernels, even if the kernel only reads or writes a subset of the data. If the data written to by the kernel is not also



```
void loop_fun(double *ZFALLSINK, double *ZQXN, double *ZRDTGDP,
    double *ZPFPLSX, int NPROMA, int KLON, int KLEV, int i, int k) {
  for (int j = 0; j < NPROMA; j++)
    ZPFPLSX[KLON*k + KLON *(KLEV+1)*i + j] =
        ZFALLSINK[KLON*i + j] * ZQXN[KLON*i + j] * ZRDTGDP[j];
}
void fun(double *ZFALLSINK, double *ZQXN, double *ZRDTGDP, double *ZPFPLSX,
    int NCLV, int NPROMA, int KLON, int KLEV, int i, int k) {
  //...
  loop_fun(ZFALLSINK, ZQXN, ZRDTGDP, ZPFPLSX, NPROMA, KLON, KLEV, i, k);
  //...
}
```

```
void fun_gpu(double *ZFALLSINK, double *ZQXN, double *ZRDTGDP,
    double *ZPFPLSX, int NCLV, int NPROMA, int KLON, int KLEV, int i, int k) {
  //...
  double *gZQXN, *gZRDTGDP, *gZFALLSINK, *gZPFPLSX;
  cudaMalloc((void**)&gZQXN, KLON * NCLV * sizeof(double));
  cudaMalloc((void**)&gZRDTGDP, KLON * sizeof(double));
  cudaMalloc((void**)&gZFALLSINK, KLON * NCLV * sizeof(double));
  cudaMalloc((void**)&gZPFPLSX, KLON * NCLV * (KLEV + 1) * sizeof(double));
  cudaMemcpyAsync(gZQXN, ZQXN, KLON * NCLV * sizeof(double),
      cudaMemcpyHostToDevice);
  cudaMemcpyAsync(gZRDTGDP, ZRDTGDP, KLON * sizeof(double),
      cudaMemcpyHostToDevice);
  cudaMemcpyAsync(gZFALLSINK, ZFALLSINK, KLON * NCLV * sizeof(double),
      cudaMemcpyHostToDevice);
  void *args[] = {&gZFALLSINK, &gZQXN, &gZRDTGDP, &gZPFPLSX, &NPROMA,
      &KLON, &KLEV, &i, &k};
  cudaLaunchKernel((void*)loop_fun, dim3(32, 1, 1), dim3(32, 1, 1), args, 0, 0);
  cudaMemcpyAsync(ZPFPLSX, gZPFPLSX, KLON * NCLV * (KLEV + 1) * sizeof(double),
      cudaMemcpyDeviceToHost);
  // CUDA free (omitted for brevity)
  //...
}
```

Writes only to indices between
j0 = KLON*k + KLON*(KLEV+1)*i
and j0 + NPROMA - 1
GPU Kernel Extraction
Entire gZPFPLSX buffer is copied back to the host, overwriting ZPFPLSX

Figure 7: GPU kernel extraction overwriting host data.

first copied on to the GPU in its entirety, this causes garbage values to be copied back to the host, potentially overwriting existing computation results. This test case took only one trial and 43 seconds to identify the transformation instance as invalid, and all other 48 invalid instances were similarly uncovered after 1-2 fuzzing trials each. This indicates that at least 16 person-hours could have been saved in identifying this issue through the use of FuzzyFlow.

Loop Unrolling. A second custom transformation where we discover bugs using FuzzyFlow performs loop unrolling. We test a total of 19 instances of this transformation on CLOUDSC, uncovering **one instance** where semantics are altered through applying the transformation. By examining the extracted test case it is discovered that the loop being unrolled has a negative loop step, iterating from `i=4` down to `i=1` with a step of `-1`, where `i` is the iteration variable. While this means that the loop is executed 4 times, the transformation incorrectly unrolls the loop by only creating 2 loop body instances.

Write Elimination. The last faulty transformation identified is a built-in optimization in DaCe which eliminates temporary write operations between computations. We run 136 instances of this transformation on CLOUDSC through FuzzyFlow, which uncovers **one instance** where the transformation causes a change in semantics. By subsuming one computation into the other in this specific instance, the test case reveals that the transformation removes an intermediate write to a data container which was marked as part of the test cutout's system state. This means that the intermediate value is read again at a later point in the application, thus implying that the write to that data container should not be removed.

7 DISCUSSION

While the proof-of-concept implementation used to demonstrate the effectiveness of the workflow enabled by FuzzyFlow is built on top of DaCe, the core elements of the approach are not reliant on any specific optimization framework. The core techniques introduced that facilitate this localized transformation testing approach consist of three dataflow graph analyses:

- Identifying all side-effects of arbitrary, structure-altering program changes.
- Reducing the input space of extracted sub-programs through a min-cut formulation on the dataflow graph.
- Deriving input constraints on extracted test cases to reduce false positives during differential fuzzing.

These analyses are facilitated by dataflow languages through exposing true data dependencies on multiple levels (see Sec. 2.2). As such, the outlined techniques can readily be applied to other dataflow representations, such as HPVM [28], Naiad [42], or Dryad [25]. Optimization frameworks utilizing different program representations may follow the outlined workflow for localized optimization testing if their internal representation can be mapped to such a parametric dataflow representation.

7.1 Limitations

There are two classes of transformation bugs that may go undetected despite the use of a dataflow representation. Firstly, if a bug is only observed with a very small number of possible input values, probabilistic fuzzing may not detect the issue. As discussed in Sec. 5.1, coverage-guided fuzzing attempts to mitigate that risk by trying sample inputs that explore different program paths.

Secondly, if a transformation manipulates user-defined callbacks or library calls that may carry side effects, a dataflow representation does not allow us to capture all possible side effects. However, it is possible to detect their presence inside of a transformation's change set and provide adequate warnings in those situations (see Sec. 3.1).

8 RELATED WORK

Almost half a century's worth of research has been dedicated to proving the validity of compilers [12]. Several techniques have been developed with the goal of verifying the transformations made by an optimizing compiler, and there are different approaches for reducing programs to smaller test cases for specific situations.

Compiler Optimization Verification. Works on verifying the preservation of semantics through compilers and their optimizing transformations have taken several different approaches. Some, such as CompCert [32, 33] or CakeML [29], take the approach of verifying the compiler as a whole, guaranteeing that any transformations and translations performed preserve program semantics. Others observe the compilation or optimization process and check the program itself for equivalence before and after each separate optimization pass in a process called translation verification [47–49]. PolyCheck [3] augments programs with light-weight checker codes to automatically verify loop-based transformations of affine programs. While these checker code acts similar to program cutouts in avoiding whole-program testing, data-centric program cutouts allow testing for arbitrary programs outside the affine space. VOC [73, 74] and a translation verification infrastructure proposed by Necula [43] use simulation relations between the IR of a program before and after optimization steps to infer whether the two are semantically equivalent. Rival [53] follows a similar approach, representing changes in the IR with symbolic transfer functions, and in CoVaC [66], equivalence checks are similarly performed with a comparison system between the transformed and original version of a program. Morpheus [38] and the verification framework VeriF-OPT [37] use a formal specification language to express transformations as rewrites on control flow graphs with side conditions to verify individual transformations. Farzan and Nicolet [18] formally verify the correctness of parallelization for read-only loop-nest by looking for multi-dimensional homomorphisms between loop-nests before and after parallelization. UC-KLEE [50] synthesizes fixed-size test inputs for C functions before and after optimization and compares their behavior with respect to exit codes and detected escaping memory locations. However, like most techniques, this is often impractical for large, scientific applications.

With our approach we remove the need for modifying the compiler or the associated translation rules by trading in formal correctness proofs for differential fuzzing. Additionally, FuzzyFlow enables the testing of arbitrary, structure altering transformations, while most formal verification approaches discussed are limited to certain program classes (e.g., affine, SCoP), or intraprocedural, structure preserving, or single-loop optimizations.

Test Case Extraction. Many test case extraction approaches and frameworks are based on a technique called delta-debugging [68, 69], where specific failure inducing changes are isolated through a binary search on the total set of changes. This is a technique employed and further developed by many works, including LLVM's Bugpoint [35], Bugfind [10], vpoiso [61], Perses [56], HDD [40], and C-Reduce [51].

However, these techniques typically answer the question of *where* a bug resides and aim to provide a minimal test case for. When it is not known if an error is present, a failure inducing input is usually not available, which makes these techniques suboptimal for asking *if* an error is present.

Program slicing [60, 63] techniques instead aim to decompose programs to obtain all statements related to a specific computation – the slicing criterium. Many approaches that slice programs statically [16, 23, 46] perform the decomposition by analyzing a program dependence graph (PDG) [19] or iterations thereof, such as value dependence graphs (VDGs) [59].

By relying on representations such as PDGs, most static slicing approaches need to perform overly conservative side effect analyses that may lead to large reductions. Through the use of parametric dataflow representations we are able to construct much smaller, side effect free program cutouts, with the added benefit of test cases remaining generalizeable to different inputs and input sizes.

Fuzzing. The body of research around fuzzing, particularly in information security, is enormous [36], with many well-known and established fuzzing frameworks like AFL [67] and AFL++ [20], funfuzz [41], or BFF [55]. The technique is also applied to testing and verifying compiler correctness [7], for example in Csmith [64], which employs black-box fuzzing to generate random test cases for C compilers to trigger unwanted behavior. Linding [34] and Sheridan [54] use a similar technique to generate random C programs for exposing inconsistencies in C compilers.

By extracting small, side effect free program cutouts to test the implications of a transformation, this allows us to build on the years of experience in fuzzing and the contributions made by each of these approaches.

9 CONCLUSION

We develop an approach that leverages parametric dataflow representations to extract side effect free program *cutouts* to capture the changes made by program transformations and test them for correctness. By solving a minimum s-t cut graph problem on the dataflow graph, we minimize the number of possible input configurations for extracted cutouts to speed up testing and improve test coverage. Using gray-box differential fuzzing, program transformations are tested under various different program conditions to uncover even input dependent faulty behavior. With a reference implementation built on the optimization framework DaCe, we demonstrate how this technique enables up to 528 times faster program optimization testing in real-world applications when compared to traditional techniques. The fully reproducible, minimal test cases with fault-inducing inputs generated when a transformation bug is found additionally allow debugging of optimizations for supercomputer-scale applications on consumer workstations.

This novel transformation test case extraction approach opens the door for further advancements in both automatic program optimization testing and debugging. By applying fuzzing to the structure of program cutouts a transformation is applied to, or the parameters for a given transformation, such as the tile size in a tiling optimization, transformations can be tested under even more varying conditions. Additionally, the properties of dataflow representations in program cutouts may be exploited to further assist in transformation debugging, by highlighting exactly at which point along the dataflow path values begin to diverge after a faulty transformation was applied. We have demonstrated how the techniques discussed in this paper enable more interactive optimization testing and debugging workflows, allowing performance engineers to save costly person-hours and distributed computing resources.

ACKNOWLEDGMENTS

This project received funding from the European Research Council (ERC) grant PSAP, grant agreement No. 101002047, and the European Union's Horizon Europe programme DEEP-SEA, grant agreement No. 955606. P.S., T.B.N., and A.N.Z. are supported by the Swiss National Science Foundation ('Ambizione' Project No. 185778, 'QuaTrEx' Project No. 209358).

REFERENCES

[1] Måns Ivar Andersson and Stefano Markidis. 2023. A Case Study on DaCe Portability & Performance for Batched Discrete Fourier Transforms. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. ACM, New York, NY, USA, 55–63. https://doi.org/10.1145/3578178.3578239

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2019. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (5 2019), 1–39. https://doi.org/10.1145/3182657

[3] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. PolyCheck: dynamic verification of iteration space transformations on affine programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 539–554. https://doi.org/10.1145/2837614.2837656

[4] Tal Ben-Nun, Johannes De Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* (2019). https://doi.org/10.1145/3295500.3356173

[5] Tal Ben-Nun, Linus Groner, Florian Deconinck, Tobias Wicky, Eddie Davis, Johann Dahm, Oliver D. Elbert, Rhea George, Jeremy McGibbon, Lukas Trumper, Elynn Wu, Oliver Fuhrer, Thomas Schulthess, and Torsten Hoefler. 2022. Productive Performance Engineering for Weather and Climate Modeling with Python. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14. https://doi.org/10.1109/SC41404.2022.00078

[6] Maciej Besta and Torsten Hoefler. 2022. Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis. (5 2022). http://arxiv.org/abs/2205.09702

[7] A.S. Boujarwah and K. Saleh. 1997. Compiler test case generation methods: a survey and assessment. *Information and Software Technology* 39, 9 (1 1997), 617–625. https://doi.org/10.1016/S0950-5849(97)00017-7

[8] Ian Briggs and Pavel Panchekha. 2022. Choosing mathematical function implementations for speed and accuracy. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 522–535. https://doi.org/10.1145/3519939.3523452

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 209.

[10] J. M. Caron and P. A. Darnell. 1990. Bugfind: a tool for debugging optimizing compilers. *ACM SIGPLAN Notices* 25, 1 (1 1990), 17–22. https://doi.org/10.1145/74105.74106

[11] Ines Chami, Bryan Perozzi, Christopher Ré, Kevin Murphy, and Sami Abu-El-Haija. 2022. *Machine Learning on Graphs: A Model and Comprehensive Taxonomy*. Technical Report. 1–64 pages.

[12] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2021. A Survey of Compiler Testing. *Comput. Surveys* 53, 1 (1 2021), 1–36. https://doi.org/10.1145/3363562

[13] GB Dantzig and DR Fulkerson. 1955. On The Max Flow Min Cut Theorem of Networks.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. http://arxiv.org/abs/1810.04805

[15] Jack Edmonds and Richard M Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM (JACM)* 19, 2 (1972), 248–264.

[16] Michael D Ernst. 1994. *Practical fine-grained static slicing of optimized code*. Technical Report.

[17] European Centre for Medium-Range Weather Forecasts. 2003. CLOUDSC cloud microphysics scheme. https://github.com/ecmwf-ifs/dwarf-p-cloudsc.

[18] Azadeh Farzan and Victor Nicolet. 2019. Modular divide-and-conquer parallelization of nested loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 610–624. https://doi.org/10.1145/3314221.3314612

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (7 1987), 319–349. https://doi.org/10.1145/24039.24041

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL ++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. https://github.com/AFLplusplushttps://www.usenix.org/conference/woot20/presentation/fioraldi

[21] Lester Randolph Ford and Delbert R Fulkerson. 1956. Maximal flow through a network. *Canadian Journal of Mathematics* 8 (1956), 399–404.

[22] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 2015-June. ACM, New York, NY, USA, 336–345. https://doi.org/10.1145/2737924.2737979

[23] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12, 1 (1 1990), 26–60. https://doi.org/10.1145/77606.77608

[24] Intel. 2003. Intel oneAPI Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html.

[25] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[26] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. *Proceedings of Machine Learning and Systems* (2021).

[27] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 943–948. https://doi.org/10.1109/ASE.2017.8115709

[28] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vol. 53. ACM, New York, NY, USA, 68–80. https://doi.org/10.1145/3178487.3178493

[29] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

[30] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[31] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[32] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *ACM SIGPLAN Notices* 41, 1 (1 2006), 42–54. https://doi.org/10.1145/1111320.1111042

[33] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (7 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[34] Christian Lindig. 2005. Random testing of C calling conventions. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*. ACM Press, New York, New York, USA, 3–12. https://doi.org/10.1145/1085130.1085132

[35] LLVM Compiler Infrastructure. 2003. LLVM bugpoint tool: design and usage. https://llvm.org/docs/Bugpoint.html.

[36] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (11 2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[37] William Mansky, Dennis Griffith, and Elsa L. Gunter. 2014. Specifying and Executing Optimizations for Parallel Programs. *Electronic Proceedings in Theoretical Computer Science* 159 (8 2014), 58–70. https://doi.org/10.4204/EPTCS.159.6

[38] William Mansky, Elsa L. Gunter, Dennis Griffith, and Michael D. Adams. 2016. Specifying and executing optimizations for generalized control flow graphs. *Science of Computer Programming* 130 (11 2016), 2–23. https://doi.org/10.1016/j.scico.2016.06.003

[39] William M McKeeman. 1998. Differential Testing For Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[40] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. ACM, New York, NY, USA, 142–151. https://doi.org/10.1145/1134285.1134307

[41] Mozilla Security. 2017. funfuzz. https://github.com/MozillaSecurity/funfuzz.

[42] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[43] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation - PLDI '00*. ACM Press, New York, New York, USA, 83–94. https://doi.org/10.1145/349299.349314

[44] George C. Necula and Peter Lee. 1998. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices* 33, 5 (5 1998), 333–344. https://doi.org/10.1145/277652.277752

[45] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. 2018. Sampled Dense Matrix Multiplication for High-Performance Machine Learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 32–41. https://doi.org/10.1109/HiPC.2018.00013

[46] Karl J. Ottenstein and Linda M. Ottenstein. 1984. The program dependence graph in a software development environment. *ACM SIGPLAN Notices* 19, 5 (5 1984), 177–184. https://doi.org/10.1145/390011.808263

[47] A. Pnueli, O. Shtrichman, and M. Siegel. 1998. The Code Validation Tool (CVT). *International Journal on Software Tools for Technology Transfer (STTT)* 2, 2 (12 1998), 192–201. https://doi.org/10.1007/s100090050027

[48] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. 151–166. https://doi.org/10.1007/BFb0054170

[49] Amir Pnueli and Anna Zaks. 2006. Translation Validation of Interprocedural Optimizations. In *4th International Workshop on Software Verification and Validation (SVV 2006)*.

[50] David A. Ramos and Dawson R. Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *International Conference on Computer Aided Verification*. 669–685. https://doi.org/10.1007/978-3-642-22110-1{_}55

[51] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[52] Martin C Rinard. 2003. Credible Compilation. (2003).

[53] Xavier Rival. 2004. Symbolic transfer function-based approaches to certified compilation. *ACM SIGPLAN Notices* 39, 1 (1 2004), 1–13. https://doi.org/10.1145/

982962.964002

[54] Flash Sheridan. 2007. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience* 37, 14 (11 2007), 1475–1488. https://doi.org/10.1002/spe.812

[55] Software Engineering Institute – Carnegie Mellon University. 2010. CERT BFF. https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=507974.

[56] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering*, Vol. 2018-January. ACM, New York, NY, USA, 361–371. https://doi.org/10.1145/3180155.3180236

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*. http://arxiv.org/abs/1706.03762

[58] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. (10 2017). http://arxiv.org/abs/1710.10903

[59] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. 1994. Value dependence graphs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*. ACM Press, New York, New York, USA, 297–310. https://doi.org/10.1145/174675.177907

[60] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[61] David B. Whalley. 1994. Automatic isolation of compiler errors. *ACM Transactions on Programming Languages and Systems* 16, 5 (9 1994), 1648–1659. https://doi.org/10.1145/186025.186103

[62] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (1 2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

[63] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2 (3 2005), 1–36. https://doi.org/10.1145/1050849.1050865

[64] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*. ACM Press, New York, New York, USA, 283. https://doi.org/10.1145/1993498.1993532

[65] Zhongming Yu, Guohao Dai, Guyue Huang, Yu Wang, and Huazhong Yang. 2021. Exploiting Online Locality and Reduction Parallelism for Sampled Dense Matrix Multiplication on GPUs. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, Vol. 2021-October. IEEE, 567–574. https://doi.org/10.1109/ICCD53106.2021.00092

[66] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *FM 2008: Formal Methods*. Springer Berlin Heidelberg, Berlin, Heidelberg, 35–51. https://doi.org/10.1007/978-3-540-68237-0{_}5

[67] Michal Zalewski. 2013. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[68] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software Engineering Notes* 24, 6 (11 1999), 253–267. https://doi.org/10.1145/318774.318946

[69] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[70] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Guillermo Indalecio Fernández, Timo Schneider, Mathieu Luisier, and Torsten Hoefler. 2019. A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* (2019). https://doi.org/10.1145/3295500.3357156

[71] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2021. NPBench: A Benchmarking Suite for High-Performance NumPy. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/3447818.3460360

[72] Alexandros Nikolaos Ziogas, Grzegorz Kwasniewski, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2022. Deinsum: Practically I/O Optimal Multi-Linear Algebra. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15. https://doi.org/10.1109/SC41404.2022.00030

[73] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. 2002. VOC: A Translation Validator for Optimizing Compilers. *Electronic Notes in Theoretical Computer Science* 65, 2 (4 2002), 2–18. https://doi.org/10.1016/S1571-0661(04)80393-1

[74] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. 2005. Translation and Run-Time Validation of Loop Transformations. *Formal Methods in System Design* 27, 3 (11 2005), 335–360. https://doi.org/10.1007/s10703-005-3402-z

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.8071953

ARTIFACT IDENTIFICATION

In this paper we present a framework – FuzzyFlow – which can extract minimal test cases from real-world applications based on a set of syntactic program transformations. These minimal test cases (program *cutouts*) can be used to test program transformations in isolation under many different input configurations using differential gray-box fuzzing. We demonstrate the use of this framework in a series of four case studies. The case studies demonstrate:

(1) A technique used to minimize the size of the input configuration space to extracted program cutouts (Case Study 1)
(2) How this approach can reduce test cases from distributed applications to single-node cutouts that can be run with a single rank (Case Study 2)
(3) How FuzzyFlow can be used to automatically uncover bugs in existing transformations in the DaCe optimization framework (Case Study 3)
(4) How efficiently FuzzyFlow detects bugs in custom program transformations in a large, real-world climate simulation application (Case Study 4)

We run our case studies on a single computer with consumer hardware. The machine is equiped with a 10-core Intel i9-7900X at 4.5 GHz, 32 GB of memory, and an NVIDIA GTX 1080Ti GPU. We use Python 3.9 with a development build of DaCe 0.14.2 (https://github.com/spcl/dace/tree/2e0abf8f3af67befaa45ec1461ad6f8590180b6d), GCC 10.3, and AFL++ 4.05c (https://github.com/AFLplusplus/AFLplusplus/releases/tag/4.05c).

The computational artifact provides a Docker image and build script thereof. The provided Docker image contains the version of FuzzyFlow used for our case studies, as well as all necessary dependencies with the corresponding versions used for the reported experiments. In addition to that, all experiment files used for the case studies (i.e., the programs that were optimized, as well as the corresponding optimizations) are included in this Docker image, together with Bash and Python scripts that can be used to run each experiment workflow for each reported case study. As such, the provided artifact can be used to fully reproduce all results reported in Section 6 (Case studies) of our paper.

REPRODUCIBILITY OF EXPERIMENTS

The artifact (Docker image) provides four scripts that can be used to run each of the Case Study experiment workflows defined before in its entirety, consequently reproducing all of the results reported in Section 6 of the paper. With the hardware architecture described before, the expected execution time to run all experiment workflows is estimated to be around 10-12 hours. We describe each of the four experiment workflows in detail in the following and how they can be used to reproduce the results reported in the paper.

Workflow 1 (Paper Section 6.1): This workflow reproduces the case study 'Minimizing Input Configurations'. In this case study we test a series of loop vectorization optimizations (built-in optimizations in the DaCe framework) on the encoder layer from the natural language model (Transformer) BERT. The input parameters used in the paper are as follows:

- N = 1024
- B = 8
- H = 16
- SM = 512
- emb = 4096
- P = 64

We report the following insights in the paper:

(1) The full application takes **12.1 seconds to run** with the listed input parameters while accelerating BLAS operations with Intel MKL (median over 10 runs).
(2) For a specific instance of the loop vectorization optimization, FuzzyFlow is able to **reduce the input configuration by 75%**.
(3) With the extracted and minimized cutout, AFL++ is able to run an average of **43.7 fuzzing trials per second** (tps).
(4) Considering the original application runtime of 12.1 seconds, which would lead to 0.083 tps, this corresponds to around **528 times faster testing**.

A single Bash script in the provided artifact can be used to reproduce each of these reported numbers. The script first runs the full application 10 times and reports the median runtime (1). The script then extracts and minimizes a cutout for the reported vectorization optimization, reporting the input configuration size before and after minimization together with a number indicating by how many percent the configuration was reduced (2). Finally, the script uses AFL++ to fuzz the extracted cutout and reports the number of fuzzing trials per second achieved (3 and 4).

Workflow 2 (Paper Section 6.2): This workflow reproduces the case study 'From Multi-Node to Single-Node'. In this case study, we demonstrate how using the cutout extraction method employed by FuzzyFlow, testing program transformations in a distributed, multi-node setting, the resulting test cases can effectively reduce tests to single-node problems, eliminating the need for multiple nodes and communication. Specifically, we extract a test case for an optimization of the Sampled Dense-Dense Matrix Multiplication (SDDMM) performed by forward propagation in a vanilla attention graph neural network, designed to run in a distributed setting using MPI. Since the optimization does not make any changes to MPI communication calls, the resulting test case can be run without communication on a single rank/node.

A Bash script in the provided artifact automatically extracts a test case for the discussed optimization and generates the corresponding C++ code. This can be used to verify that the generated C++ code does not contain any MPI communication and may be run on a single node, as shown in the (simplified) code from Figure 6 in the paper.

Workflow 3 (Paper Section 6.3): This workflow reproduces the case study 'NPBench'. In this case study we use a set of micro-benchmarks (52) from the benchmark suite NPBench (https://github.com/spcl/npbench) to test all of the built-in optimizations that the DaCe framework can apply to those programs. Some types of transformations that require special hardware like FPGAs were omitted. We use FuzzyFlow to test a total of **3,280 transformation instances** on the entire benchmark set. We report **6 transformations** from the DaCe framework that contain bugs according to the resulting failing test cases.

A Bash script in the provided artifact can be used to run FuzzyFlow on the tested benchmarks and DaCe transformations. This generates all 3,280 resulting test cases and tests them with differential fuzzing. All failing test cases are reported and through performing a manual triage the 6 transformation bugs reported in Table 2 of Section 6.3 can be found and reconstructed.

Workflow 4 (Paper Section 6.4): This workflow reproduces the case study 'Optimizing Weather Forecasts'. In this case study we test 2 custom transformations and one built-in optimization from the DaCe framework on a real-world application. The custom transformations were taken from a group of engineers that were in the process of optimizing the cloud microphysics scheme (CLOUDSC) of the European Centre for Medium-Range Weather Forecasts' (ECMWF) Integrated Forecasting System (IFS). The engineers wrote custom transformations to automatically extract GPU kernels and to perform loop unrolling. We use FuzzyFlow to test each instance of these custom transformations on the CLOUDSC application – specifically on different versions of CLOUDSC taken from different stages of their optimization workflow. Similarly, we test each instance of a built-in optimization from the DaCe framework that tries to eliminiate intermediate writes on CLOUDSC.

We report the following insights in the paper:

(1) 62 instances of the GPU kernel extraction transformation were tested, **48 of which** changed program semantics.
(2) Testing a single instance of a GPU kernel extraction transformation took only **43 seconds**
(3) Most semantics altering transformation instances of the GPU kernel extraction were uncovered after only 1 or 2 fuzzing trials.
(4) 19 instances of the loop unrolling transformation were tested, **one of which** changed program semantics.
(5) 136 instances of the built-in write elimination transformation were tested, **one of which** changed program semantics.

A Bash script in the provided artifact can be used to run FuzzyFlow on CLOUDSC for all three of the listed transformations, reproducing each of these insights. The script first tests the GPU kernel extraction transformation on CLOUDSC, extracting a minimal test case and demonstrating that finding a single bug only takes 43 seconds (2). The script then tests each instance of the loop unrolling transformation a version of CLOUDSC, demonstrating that from all tested 19 instances, the failing test case can be found (after manual triage) (4). Finally, the script tests each instance of the write elimination transformation on two versions of CLOUDSC. This demonstrates that all 136 instances are quickly tested individually, each with a minimal test case. Out of all tested instances, a few test cases fail testing and after performing triage, the bug can be identified and reconstructed (5).

ARTIFACT DEPENDENCIES REQUIREMENTS

To reproduce all experiments, the artifact requires an NVIDIA GPU with CUDA 12.0.x capability. The experiments may use up to 16 GB of RAM, and at least XX GB of storage space is required for caching and storing experiment data. There are no other hardware requirements, and there are no specific operating system requirements.

This artifact requires an up-to-date installation of Docker and Git to function, but does not carry any other software or library dependencies. All necessary software and libraries are handled through the use of a Docker image.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

The artifact DOI points to a GitHub repository containing everything necessary to build the artifact's Docker image. To install and deploy everything necessary for reproducing the reported experiments, proceed through the following steps:

(1) Clone the GitHub repository at the commit specified by the DOI and navigate into the the cloned directory with your terminal. Once inside the repository, make sure submodules are cloned by running `git submodule update –init –recursive` (Time requirement: < 5 minutes)
(2) Build the artifact's Docker image by running: `docker build . -t fuzzyflow` (Time requirement: 10-30 minutes)
(3) Launch an interactive Docker container from the built image by running: `docker run –rm -it fuzzyflow` (Time requirement: a few seconds to minutes)

Now inside the artifact Docker container, you may reproduce the reported experiments with four workflows (see "Reproducibility of Experiments"). Each workflow can be launched through a single Bash script:

(1) To run workflow 1, reproducing Paper Section 6.1, run `./run_workflow_01.sh` (Time requirement: < 10 minutes)
(2) To run workflow 2, reproducing Paper Section 6.2, run `./run_workflow_02.sh` (Time requirement: < 5 minutes)
(3) To run workflow 3, reproducing Paper Section 6.3, run `./run_workflow_03.sh` (Time requirement: 6-10 hours)
(4) To run workflow 4, reproducing Paper Section 6.4, run `./run_workflow_04.sh` (Time requirement: 1-1.5 hours)

All of these instructions, as well as the workflow descriptions, can also be found in the README contained in the GitHub repository under the artifact's DOI.

Note that all time requirements assume comparable hardware to what was used in the paper and do not include manual triage of test cases for false positives, as that may vary significantly and is not a core part of the experiments.