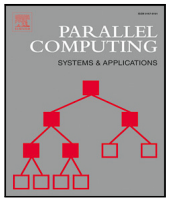
Parallel Computing 121 (2024) 103097

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco



An automated OpenMP mutation testing framework for performance optimization

Dolores Miao [a,*], Ignacio Laguna [b], Giorgis Georgakoudis [b], Konstantinos Parasyris [b], Cindy Rubio-González [a]

[a] University of California, Davis, CA, 95616, USA
[b] Lawrence Livermore National Laboratory, Livermore, CA, 94550, USA

ARTICLE INFO

Keywords:
Mutation testing
OpenMP
Performance optimization
Dynamic program analysis

ABSTRACT

Performance optimization continues to be a challenge in modern HPC software. Existing performance optimization techniques, including profiling-based and auto-tuning techniques, fail to indicate program modifications at the source level thus preventing their portability across compilers. This paper describes Muppet, a new approach that identifies program modifications called *mutations* aimed at improving program performance. Muppet's mutations help developers reason about performance defects and missed opportunities to improve performance at the source code level. In contrast to compiler techniques that optimize code at intermediate representations (IR), Muppet uses the idea of source-level *mutation testing* to relax correctness constraints and automatically discover optimization opportunities that otherwise are not feasible using the IR. We demonstrate the Muppet's concept in the OpenMP programming model. Muppet generates a list of OpenMP mutations that alter the program parallelism in various ways, and is capable of running a variety of optimization algorithms such as delta debugging, Bayesian Optimization and decision tree optimization to find a subset of mutations which, when applied to the original program, cause the most speedup while maintaining program correctness. When Muppet is evaluated against a diverse set of benchmark programs and proxy applications, it is capable of finding sets of mutations that induce speedup in 75.9% of the evaluated programs.

1. Introduction

Performance optimization continues to be a challenge in modern HPC software. The adoption of multi-core heterogeneous systems and the use of multi-process and multi-threaded programming models to fully utilize modern architectures are some of the factors that limit the ability of developers to solve performance issues; these issues can result in poor user experience, lower system throughput, limit scalability, and a waste of computational resources [1–3].

Problems with Existing Techniques. A lot of work has been proposed to identify performance issues and several tools are used in current HPC production environments to analyze the performance of applications [4–7]. However, the process of isolating performance problems and/or generating tests to identify them is still mostly a manual process. Most performance optimization techniques focus on highlighting performance hotspots in the program, but ultimately they rely on programmers to identify code modifications that fix a performance problem or improve overall performance. Other approaches are based on the concept of quantifying hardware or runtime system events [8–10], but metrics of these events do not directly relate to program performance, and these approaches do not explicitly inform the programmer how to modify the code to improve performance. Compiler optimizations improve performance usually at the intermediate representation (IR) level; however, reasoning about correctness at the IR level is much more difficult than at the source level. As a result, compiler optimizations can leave optimization opportunities on the table. Moreover, IR-level optimizations are not portable across compilers.

We could potentially solve performance problems given accurate performance models for each available platform and application. If performance models are available, we could simply check if application behavior falls into the bounds of such models. However, such an ideal mechanism is hard to realize in practice as performance models are notoriously difficult to build accurately given the complexity of the HPC software stack and underlying hardware. There are solutions to build performance models for specific aspects of the hardware and applications [11–13], but they are usually not composable and thus of little practical use in modeling an entire application and platform.

* Corresponding author.
E-mail address: wjmiao@ucdavis.edu (D. Miao).

https://doi.org/10.1016/j.parco.2024.103097
Received 3 May 2024; Received in revised form 10 July 2024; Accepted 12 August 2024
Available online 21 August 2024

Our Contributions. We present an approach based on *mutation testing* [14] to identify source code changes, or *mutations*, that (1) improve performance, and (2) help developers reason about performance at the source-code level, in contrast to IR- or assembly-level like in existing methods. Since such an approach is based on source modifications, it is portable across compilers.

Mutation testing has been proposed to identify correctness faults [14], and assumes that a syntactic change (a mutant) along with an exploration campaign of multiple mutants can help discover program defects faster than traditional methods. While some previous work has applied mutation testing to solve performance defects [15], mutation testing for performance has not been applied on parallel code and/or HPC programs. We demonstrate our approach in the OpenMP programming model, although our approach is also applicable to other HPC programming paradigms such as CUDA and OpenACC.

We implement our approach in the framework named Muppet (Mutation-Utilized Parallel Performance Enhancement Tester). First, Muppet generates a list of OpenMP mutations, which are defined as either a change in an existing OpenMP directive in the program that could change the performance of the code block that the directive targets, or adding a new OpenMP directive that introduces parallelism to existing serial code. Muppet considers only mutations that are not likely to change the correctness of the code block. Next, Muppet considers different optimization algorithms, such as delta debugging [16], Bayesian optimization (BO) [17], and decision tree optimization [18], to find a subset of mutations that, when applied to the original program, causing the highest speedup. We implement Muppet in the Clang/LLVM front-end and evaluate it in a variety of programs, including benchmark programs like the Rodinia benchmarks [19], the NAS Parallel Benchmarks [20], HPCG [21], and four scientific applications (LULESH [22], CoMD [23], CoSP2 [24], and the CLOUDSC cloud physics mini-app [25]).

In summary, our contributions are:

- We present a source-level approach that uses mutation testing to optimize HPC code. Our approach considers five classes of source mutations and applies them in OpenMP directives. *To the best of our knowledge, we are the first to explore using mutation testing to optimize OpenMP code* (Section 3).
- We design and implement our idea in the Muppet framework via the Clang/LLVM front-end. Our approach integrates Muppet with several optimization algorithms, such as delta debugging, Bayesian optimization and decision tree optimization. The output of Muppet is a set of source modifications, or mutations, that produce a maximum speedup among the explored mutations, without affecting correctness (Section 4).
- We evaluate Muppet on several benchmarks and proxy applications. We demonstrate that Muppet is capable of identifying mutations that improve performance in 75.9% of the evaluated programs, with the best speedup of 3.57x (Section 5).

Comparison to Previous Work. An earlier version of our approach was published in [26]. The approach has been expanded in this paper and a more comprehensive experimental evaluation has been conducted with new key findings. New insights include:

- A new OpenMP mutation focusing on thread affinity management has been added in Muppet: `schedule`.
- A new optimization algorithm, decision tree optimization, has been integrated into Muppet and evaluated for its efficacy.
- A set of new programs, including benchmark programs and scientific applications, is evaluated alongside programs tested in the original paper, with varying levels of code complexity. The maximum speedup discovered is higher than in the previous paper.
- In addition to comparing speedups discovered by different algorithms, we also compare the time used to run different algorithms to find these speedups. With this information, we can illustrate a more complete picture of how Muppet performs across programs from different knowledge domains, with different data structures, and with varying time complexity.
- We provide new detailed analysis of source code and associated OpenMP mutations on a set of selected programs that have performance speedup discovered.

2. Overview

In this section, we describe the philosophy of our approach, provide background information on mutation testing, and provide a simple mutation example in a matrix multiply kernel that improves performance.

2.1. Approach's philosophy

Existing approaches to isolate performance issues are difficult to use in practice. A number of performance problems can be fixed by changes in the source code; however, existing methods do not directly point to developers' source modifications that fix such issues. Compilers optimize code at the IR level but such solutions are not portable across compilers and make it harder to reason about correctness than solutions based on source modifications.

We believe that tools and techniques for performance optimization should have the following features:

- **Fine granularity detection:** tools should pinpoint, with fine granularity, the location (code line) of performance issues or potential performance improvements.
- **Guided fixes:** the approach should help programmers understand and reason about performance defects—without a good understanding, it is hard to solve the problem or avoid it in the future.
- **Automatic recommendations:** the approach should automatically suggest code modifications that improve performance or fix a performance problem.

We designed Muppet using the above criteria to identify changes in OpenMP directives that improve performance.

2.2. Mutation testing for performance

2.2.1. Challenges

The key idea of Muppet is to perform small changes in the code, called *mutations*, and use exploratory algorithms to search for cases where mutations improve performance or fix a performance problem. Mutation testing has been studied before to detect faulty programs by injecting small syntactical changes that expose correctness defects [14]. The idea of mutation testing is to generate sufficient data to expose real software defects in the code. However, it is challenging to use traditional mutation testing in isolating performance defects because the syntactic changes could create faults, i.e., breaking the semantics of the program and producing incorrect programs.

2.2.2. Our solution

Inspired by the previous work on mutation testing, we propose a different approach: *to inject only mutations that are semantically correct and do not yield an incorrect program for the purpose of exposing performance defects or speedup opportunities.* Semantically correct mutants, or *equivalent mutants*, are considered problematic for traditional mutation testing because by definition, they cannot fail the test suite, so they should be avoided to increase the effectiveness of mutation testing. In contrast, our approach explores semantically correct mutations, or a weaker form of mutations that successfully pass correctness tests, to identify any mutations that increase performance, thus indicating performance defects.

2.3. Mutation example

Here, we present a synthetic matrix-multiplication example, shown in Listing 1, that demonstrates Muppet's capabilities—when we apply Muppet, it can find a set of mutations that yields faster code execution.

Listing 1: Example code with a mutation found by Muppet that improves performance.

```
#define ARRAY_SIZE (2048)
double A[ARRAY_SIZE][ARRAY_SIZE];
double B[ARRAY_SIZE][ARRAY_SIZE];
double C[ARRAY_SIZE][ARRAY_SIZE];

int main(void) {
  // initialize array and timer setup
      omitted
  float var = 2.3f;
  #pragma omp parallel for shared(var)
// mutation adds an OpenMP directive
#pragma omp tile sizes(16,16,16)
  for (int i = 0; i < ARRAY_SIZE; ++i)
    for (int j = 0; j < ARRAY_SIZE; ++j)
      for(int k = 0; k < ARRAY_SIZE; ++k) {
        C[i][j] += var*A[i][k]*B[k][j];
      }
  // end processing omitted
}
```

Originally, the code has only the OpenMP `parallel for` directive to parallelize the loop. Muppet applies mutations to the existing OpenMP directives found in the code. Note that while Muppet only considers semantically correct mutations (and are likely to produce a correct program), it relies on existing correctness checks of the program, as shown in Section 5.1.1 for the evaluated programs. Possible mutations are shown in Fig. 2 as references. When we run Muppet on this example with delta debugging, after 20 tryouts, Muppet reports a mutation that, when applied to the program, improves performance. With BO, it takes 66 tryouts to finish the optimization process; but the mutation was reported with 11 tryouts. The identified mutation is highlighted in the source code. In this simple example, the mutation is the addition of the OpenMP tile construct, which converts the three-dimensional loop space in this program into smaller-sized "tiles" in $16 \times 16 \times 16$ increments and suggests to the compiler that each tile is to be assigned to one OpenMP thread. In the end, Muppet reports to the developer that adding this construct to the loop introduces a 18.84x speedup, from 7.116801 s to 0.377674 s.

3. Approach

3.1. Problem statement

Given an OpenMP program P with running time T, Muppet analyzes the program and generates a set of mutations, $M = \{m_1, m_2, \ldots, m_n\}$, which potentially could induce program speedup. We define the program running time for the original variant program as:

$$T = P(\emptyset)$$

We define the running time for a variant program as:

$$T' = P(M'), \text{where} \quad M' \subseteq M, \text{and} \quad accurate(P, M') = True.$$

We define the ideal minimum program running time as:

$$T_{min} = P(M_{min}),$$
$$\text{where} \quad M_{min} \subseteq M,$$
$$\text{and} \quad accurate(P, M_{min}) = True,$$
$$\text{and} \quad \forall M' \subseteq M, T' >= T_{min}$$

The goal of Muppet is find a subset of M, $M_{min'}$, with $T_{min'}$ as close to T_{min} as possible.

3.2. Tool workflow

The overall workflow of Muppet is illustrated in Fig. 1. The purposes of these modules are described below:

- *Mutation generator* analyzes the program and finds a set of source code mutations, which can potentially be applied to change the OpenMP parallelism of the program.
- *Transformer* generates a program variant with a subset of mutations found in the *Mutation generator* module.
- *Tester* runs the mutated programs from *Transformer* and tests the performance speedup and correctness of the mutated variant.
- *Optimizer* applies a user-specified optimization algorithm to find the minimum of the function $T' = P(M')$.

Next, we delve into the details of these modules, following the order as they appear in Fig. 1.

3.3. Mutation generator

The Mutation Generator module traverses the abstract syntax tree (AST) of the program, looking for source code locations that potentially can be mutated so that program parallelism is changed. The time complexity of this step is $\mathcal{O}(n)$ where n is the number of statement nodes on the AST. The mutators in Muppet focus on mutating parallel/loop OpenMP constructs such as the `parallel` directive, `for` directive, or the `parallel for` directive. All of these directives specify a source code region to be executed in parallel, but the parallelism may not be high enough to utilize all available cores for the OpenMP program. Muppet also looks for the beginning of `for` loops for both SIMD mutations. Lastly, `tiling` mutations may be added in case tiling can be performed in an inner part of a multiple-dimension parallel loop.

3.3.1. Mutation classes

There are five classes of mutations possible to apply to certain source code locations:

- **Collapse Mutations** add a `collapse` clause to a multiple-dimension `parallel for` loop. Collapse clauses may potentially improve parallelism by having more iterations, thus higher hardware thread usage, at the top level of the loop.
- **SIMD Mutations** have two forms: adding a `simd` clause to an OpenMP parallelism-related directive such as a `parallel for` loop or a `omp for` loop; or adding a `simd` directive to a `for` loop. SIMD clauses or directives hint at the compiler to check if there is a possibility to vectorize the loops and apply SIMD vectorization if possible.
- **Tiling Mutations** add `tile` directives at the top of a multiple-dimension OpenMP loop. Tile directives split the loop space into smaller-sized "tiles", and each tile is ideally only accessed by one OpenMP thread. This design can potentially improve cache locality depending on how the data within memory is accessed within the loops, and thus may also introduce performance speedup. Due to the difficulty in determining loop size at compile time, Muppet only supports setting a fixed set of differently sized tiles as different mutations. For example, we can only set the tile size as a power of 8, 16, or 32. Given the limitations, users can still see from the optimization results whether using a smaller or larger-sized tile can have a higher speedup.
- **Firstprivate Mutations** put read-only shared variables into a `firstprivate` clause for an OpenMP parallel region, so that these variables are kept a copy in each parallel thread. This is to reduce data dependency between parallel threads when these variables are accessed.

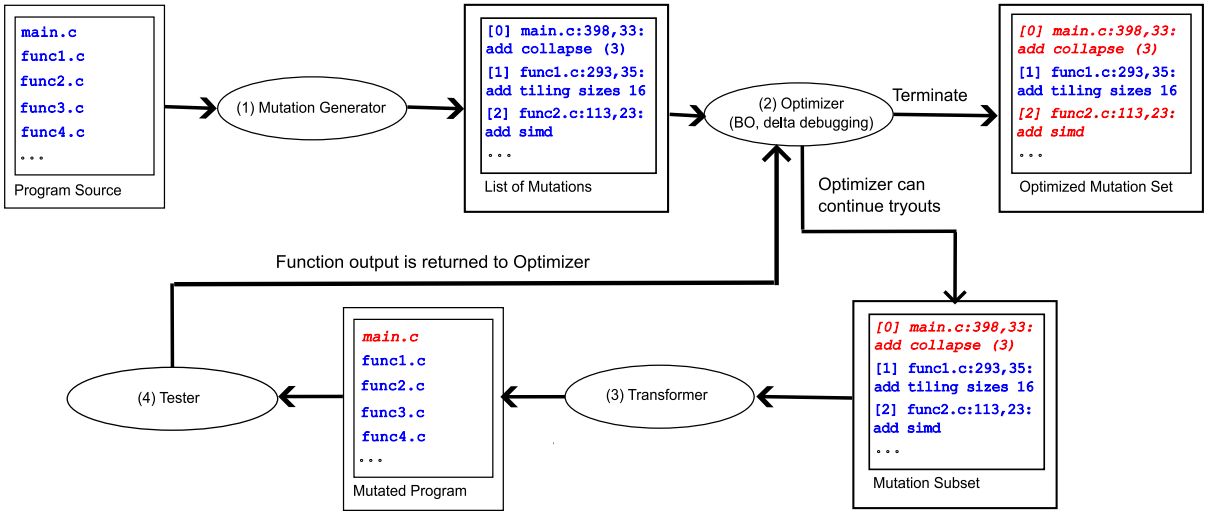


Fig. 1. The workflow of Muppet. *Red texts in italic* indicates the mutation is applied, or the source file is changed.

- **Schedule Mutations** add a `schedule` clause to an OpenMP `parallel` or `parallel for` directive. OpenMP by default statically assigns loop iterations to OpenMP threads when running a parallel block. This mutation modifies the scheduling behavior of OpenMP programs. Similar to tiling mutations, Muppet supports two different schedule mutations: dynamic- and auto-scheduling, implemented as two different mutations for each parallel block.

For all mutations, once a language construct of interest is detected, the Mutation Generator module then checks the associated source code around the current language construct. If the source code around it satisfies certain statically defined criteria (see Section 3.3.2), then unique information regarding the current mutation, such as source location, the way source code is modified with this mutation (insert before a source code location, insert after a source code location, modify a source code range), and the class of the mutation is added to the list of mutations. The algorithm for this process is shown in Algorithm 1.

Algorithm 1: The mutation generator algorithm.

```
1  Function GenerateMutations(AST):
2      M = ∅;
       // Traverse the AST.
3      foreach Statement in AST do
4          if Statement is an OpenMP directive then
5              if can add collapse mutation then
6                  M = M ∪ CollapseMutation(Statement);
7              if can add SIMD mutation then
8                  M = M ∪ SIMDMutation(Statement);
9              if can add tiling mutation then
10                 M = M ∪ TilingMutation(Statement);
11             if can add firstprivate mutation then
12                 M = M ∪ FirstprivateMutation(Statement);
13             if can add schedule mutation then
14                 M = M ∪ ScheduleMutation(Statement);
15         if Statement is a for loop then
16             if can add SIMD mutation then
17                 M = M ∪ SIMDMutation(Statement);
18             if can add tiling mutation then
19                 M = M ∪ TilingMutation(Statement);
20     return M
```

3.3.2. Criteria selection

The criteria for each class of mutation follow the syntax of OpenMP language specifications. These criteria can be expanded for any new class of mutations added. A list of criteria is below:

- Mutations should not be added in a `for` loop that has statements that change the control flow, such as a `break`, `continue` and a `return` statement. OpenMP parallel loops in existing code already follow this rule, but other standalone `for` loops may not, so condition checking is necessary.
- Tiling and SIMD mutations should not be added when the affected OpenMP block contains specific OpenMP directives such as a `critical`, `barrier`, or a `master` directive.
- There should not be OpenMP blocks inside SIMD directives, otherwise such mutations should not be added.
- In general, there should only be SIMD directives inside a SIMD directive, and no other directives are allowed.

We illustrate OpenMP mutations that can be applied to in the previously shown *matmul* example in Fig. 2. The one that shows the highest speedup in *matmul* is the tiling mutation.

3.4. Optimizer

Once a list of mutations is generated, it is exported to the Optimizer. This module runs an optimization algorithm specified by the end user to find the minimum point of $T' = P(M')$. During the optimization process, it finds specific points on the $T' = P(M')$ function by selecting a subset of mutations, sending these mutations to the Transformer and Tester module, and receiving T' from the Transformer and Tester module once the mutated program has finished execution and running time statistics are collected.

Muppet supports three optimization algorithms: delta debugging [16], Bayesian Optimization [17], and decision tree optimization [18]. The goal of all three algorithms, albeit vastly different in implementation, is to find the subset of source mutations that would introduce maximum speedup. The inclusion in Muppet of a variety of algorithms shows how algorithms with vastly different original purposes can solve the same problem in different ways with varying efficacy. Muppet can also be extended to run other optimization algorithms such as differential evolution or simulated annealing. The details for each algorithm and how they are adapted to Muppet are detailed below.

3.4.1. Delta debugging

Delta debugging (DD) is an algorithm that was originally developed as a software testing algorithm to isolate bugs inside a program, which is then adapted into finding speedup in program variants in previous work such as Precimonious [27] with regards to precision tuning. We

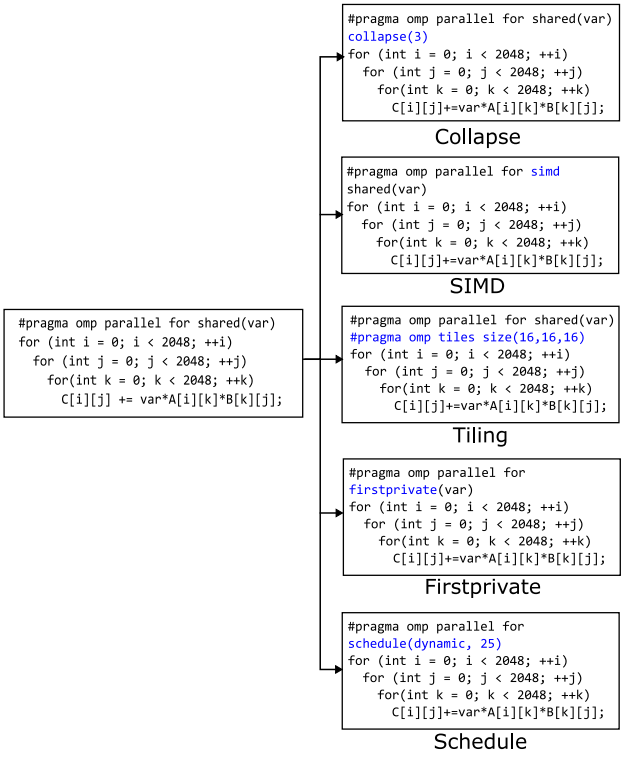


Fig. 2. Classes of mutations in Muppet.

follow the LCCSearch algorithm in [27], where a change set in our adaptation of the algorithm is defined as the set of mutations that are applied to the original program, and the outputs are a minimal change set which causes speedup.

To illustrate how delta debugging is adapted to Muppet, we show two simple examples in Fig. 3. The change set is initially assigned as the set of all mutations. Then Muppet starts testing mutated program variants with all mutations, with the first half of mutations, and with the second half of mutations. In (a), the second half of mutations cause speedup, thus the second half of mutations become the new change set, and then this new change set is split in two halves and test their performance speedup respectively. In (b), none of the halves cause speedup, thus Muppet split the mutations into 4 parts instead of 2 and test each part and its complement set, respectively, as shown in Round 3. If a single part among these 4 parts causes speedup, then similar to (a), that part becomes the new change set and is split again in two for further testing. If a complement set (3 parts in this case) causes speedup, as shown in the figure, this means that excluding that one part improves performance, thus the change set becomes the remaining 3 parts for further performance testing, as shown in Round 4. The algorithm finishes running when every part in the algorithm is a single mutation and cannot be split further.

The time complexity of delta debugging is $O(n \cdot log(n))$ where n is the number of possible mutations in the average case; in the worst case, the time complexity is $O(n^2)$.

3.4.2. Bayesian optimization

Bayesian Optimization (BO) is a common optimization algorithm that approximates a computationally expensive function, such as the programs tested in Section 5. In Muppet, Gaussian processes are used as surrogate models to approximate program time characteristics, an acquisition function is used to predict the next input to be tested for performance, and the result of the next input (running time) is then sent back to improve the surrogate model. BO does not have the assumption of the function forms, which makes it an appropriate algorithm to use in Muppet.

There is one difficulty in adapting BO to Muppet: BO requires the function to be in the form of $y = f(x)$ where x is a list of number inputs, while y is a floating-point output (in our case the run time of the program). Meanwhile the input parameter of the function to be optimized, $T' = P(M')$, is a subset of mutations. Therefore we optimize $T' = P(Mb')$ instead where:

$$Mb' = \{mb_1, mb_2, \ldots, mb_n\}$$

$$mb_i = \begin{cases} 1, & \text{if} \quad m_i \in M' \\ 0, & \text{otherwise} \end{cases}$$

In this way, we assign 0 or 1 for each element in the set of mutations according to whether the mutation is in the subset. For example, if $M = \{m_1, m_2, m_3, m_4\}$, $M' = \{m_1, m_3\}$, then $Mb' = \{1, 0, 1, 0\}$. The converted list of 0 and 1 can be accepted by BO as input parameters.

The computational complexity of Bayesian optimization is $O(n^3)$ where n is the number of mutations, however, some methods reduce the computation time, as shown in [28].

3.4.3. Decision tree optimization

Similar to BO, decision tree optimization (FO) also uses a surrogate model to approximate an expensive function, and an acquisition function to estimate the next input for running time evaluation, and the running time output is also sent back to improve the surrogate model. However, in this case, a decision tree regression model is used instead. We decide to use random forest [29] as the surrogate model here. Also similar to BO, FO optimizes $T' = P(Mb')$ function by assigning 0 or 1 for each mutation in the mutation set.

The computational complexity of decision tree optimization using random forest is $O(n \cdot log(n) \cdot d \cdot k)$ where n is the number of points in the training set, d is the number of mutations, and k is the number of decision trees in the forest.

3.5. Transformer and tester

The Transformer and Tester modules read the list of mutations from the Optimizer module, mutate the program into a variant, and run the variant to see if there is any speedup while maintaining the correctness of the program.

3.5.1. Compilation and conflicts checks

Even though there are already criteria placed in the Mutation Generator module for each mutation class to ensure that all mutations generated are syntactically correct, there are still situations where two mutations, when applied to the same programs at the same time, cause conflicts between them. If Muppet lets these conflicts pass without checking during the transformer phase, it would cause a large number of mutated program variants that do not compile. Thus, to save execution time, when the Transformer module traverses the program, it also statically checks and circumvents certain conflicts. These conflict checks can also be customized in the case where new classes of mutations are implemented or new conflicts are discovered during testing. Existing conflict checks in Muppet are listed below:

- An active SIMD directive mutation should not be inside an active collapse or tiling mutation, and will be discarded.
- An active tiling directive mutation should not be inside an active SIMD mutation, and will be discarded.
- Every variable inside an OpenMP parallel region is checked. If the variable is read-only, and not in a list of existing `private` variables, then it will be included in the list of variables in the firstprivate mutation.

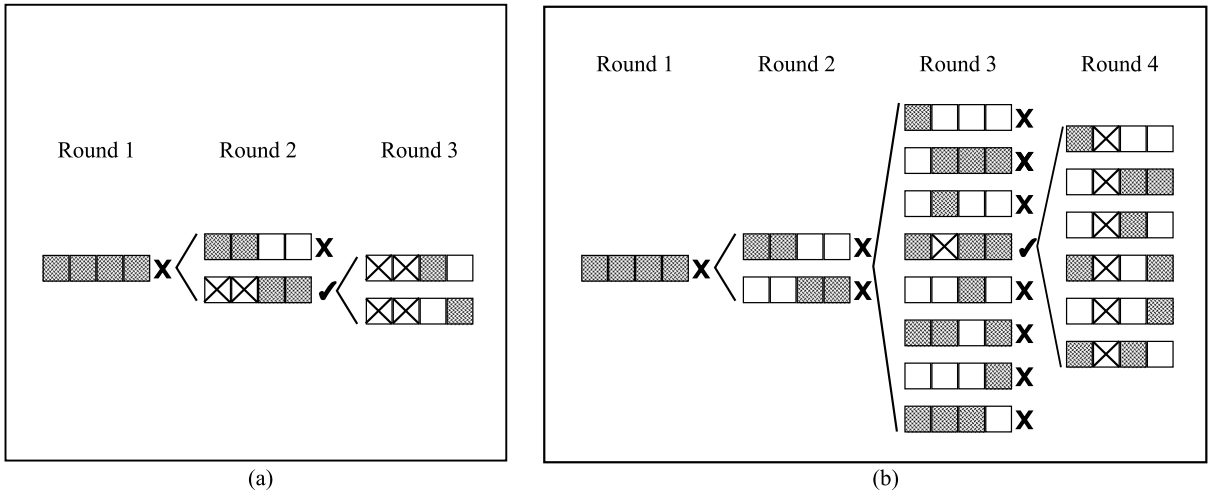


Fig. 3. Two simple examples illustrating the delta debugging algorithm.

4. Implementation details

4.1. Tools used to implement Muppet

Muppet is implemented with a variety of programming languages and toolsets. The Mutation Generator and the Transformer modules are implemented via Clang plugins. Given the nature of our work on source-level OpenMP mutations, Muppet is compatible with programs compiled with any C/C++ compiler as long as it supports OpenMP 5.0.[1] There are a few mechanisms in the Clang compiler architecture that are capable of performing source-to-source code transformation besides Clang plugins, such as libtooling and libclang. Sending Clang plugin calls as compiler parameters to the build system ensures that all source files are processed by the Mutation Generator and the Transformer modules. Muppet only requires minimal changes to the build scripts for it to work on new programs, which is described in Section 4.3.

The Optimizer and Tester modules, and the overarching framework managing the communication between modules, on the other hand, are implemented in Python 3.10. This is done to leverage the existence of a mature set of Python optimization modules such as scikit-optimize [30].

4.2. Modular extension

Since Muppet uses a modular approach, each of the four modules shown in Fig. 1 can be replaced to implement an analogous functionality. Optimization algorithms can be replaced, as mentioned in Section 3.4. Even language support can be extended; while currently Muppet targets C/C++ programs with OpenMP language constructs, it is possible to target FORTRAN programs by rewriting the Mutation Generator and Transformer modules with a source-to-source FORTRAN compiler such as ROSE [31].

4.3. Making Muppet work with your own program

For better management of programs, Muppet calls a customized version of the FAROS build system [32]. Muppet accepts a YAML config file that contains program entries, with commands for a variety of functionalities such as analyzing, transforming, building and running the specified program. With FAROS, it is easy to add new programs to perform OpenMP mutation testing for speedup by simply adding new entries into the YAML config file.

4.3.1. Entries and correctness

An example entry for a locally stored simple matrix multiplication program is shown in Listing 2. It sets up commands for each step used in Muppet, such as building, calling plugins for mutations, running the program, extracting running time statistics from program output, and cleaning. Note that the `call_plugin` part of the YAML file was not originally a part of FAROS and is an addition of Muppet to FAROS to analyze and transform program source code. In addition to the YAML file, the only changes required for the *matmul* source code is (a) modify the build scripts (Makefile targets `func_analysis` and `trans_mutations` in this case) so that it accepts parameters for calling the Clang plugins, and (b) add correctness check code that parses program output and determines if the mutated program variant still runs correctly.

Listing 2: YAML config file for *matmul*.

```
matmul: fetch: 'cp -r ../../../extra/matmul .'
  build_dir: 'matmul'
  build: {
    omp: ['make CC=clang++ OPT_LEVEL=3 OMP=1'],
  }
  call_plugin: {
    analysis: ['make func_analysis OMP=1'],
    mutate: ['make trans_mutations OMP=1'],
  }
  copy: ['matmul' ]
  bin: 'matmul'
  run: './matmul'
  input: ''
  measure: 'Work consumed (\d+\.\d+) seconds'
  clean: 'rm -r *.*; cp ../../../../extra/matmul/*.* .'
```

4.4. Customizing Muppet runtime parameters

User can select between delta debugging, Bayesian optimization and decision tree optimization when using Muppet. Bayesian optimization and decision tree optimization are implemented with `gp_minimize` and `forest_minimize` functions in scikit-optimize, while delta debugging is implemented from scratch, adapting the algorithm described in Precimonious [27], since it has no publicly available Python implementations.

[1] By disabling certain classes of mutations such as tiling, Muppet is compatible with programs built with compilers that only support lower versions of OpenMP such as 4.5, although the speedup discovered would potentially be lower.

Table 1
Problem size and running time for evaluated programs.

(a) Benchmarks (Rodinia, NPB-CPP, HPCGs)

Program	Parameters	Min. Time (s)	Avg. Time (s)
backprop	16777216	13.270	13.506
cfd	fvcorr.domn.193K	18.527	18.535
b+tree	file mil.txt command command.txt	1.536	1.539
heartwall	test.avi 20 14	2.259	2.266
hotspot	1024 1024 16384 14 temp_1024 power_1024	3.896	3.921
hotspot3D	512 8 1000 power_512 × 8 temp_512 × 8	2.726	2.737
kmeans	kdd_cup	1.727	1.752
lavaMD	-cores 14 -boxes1d 32	4.177	4.191
leukocyte	5 14 testfile.avi	1.481	1.549
lud	-n 14 -s 3200	5.638	5.847
myocyte	1000 500 1 14	5.186	5.221
nn	filelist.txt 10000 30 90	1.825	1.835
nw	32000 10 14	1.332	1.346
particlefilter	-x 512 -y 512 -z 100 -np 10000	2.936	2.991
pathfinder	1000000 100	2.992	3.066
srad	1024 1024 0 127 0 127 1 0.5 100	3.443	3.476
streamcluster	10 20 256 65536 65536 1000 none output.txt 14	10.780	11.066
FT	CLASS=A	1.257	1.262
LU	CLASS=A	4.933	4.956
MG	CLASS=A	3.617	3.704
SP	CLASS=A	31.526	31.675
BT	CLASS=A	42.574	42.590
CG	CLASS=B	22.512	22.735
EP	CLASS=B	6.244	6.248
HPCG	96 96 96	15.548	15.620

(b) Scientific applications

Program	Parameters	Min. Time (s)	Avg. Time (s)
LULESH	-i 1500 -s 35	11.346	11.740
CoMD	-e -i 1 -j 1 -k 1-x 20 -y 20 -z 20	2.304	2.418
CoSP2	–hmatName hmatrix.1024.mtx–N 12288 –M 16384	4.266	4.312
CLOUDSC	14 100 4 (500 iterations)	8.160	8.187

Since the running time for each program run may have variability that should not be counted as speedup, to reduce the impact of such variability affecting the reported speedup, users can customize Muppet parameters to change how it measures running time. The *times* parameter instructs Muppet to run a specified number of repetitions for each variant, and collect running times for each run; the *shuffle* switch, only available for delta debugging, randomly shuffle the order of mutations so that the delta debugging algorithm partitions these mutations differently each time (users can still specify the same random number generator seed for the same shuffle result). Lastly, users can choose between using the minimum running time in all repetitions, or using the average running time, as the fitness function for all optimization algorithms. Our evaluation of Muppet uses some of these parameters which are discussed later in Section 5.1.1.

5. Experimental evaluation

This evaluation answers the following research questions:

RQ1 Does Muppet discover source code mutations that induce speedup for OpenMP programs, and how does it perform with different algorithms?

RQ2 How does Muppet compare when using different optimization algorithms, with limited time budget/tryouts?

RQ3 What kind of mutations does Muppet discover that cause significant speedup?

5.1. Evaluation setup

5.1.1. Benchmarks

We use a variety of C/C++ OpenMP programs to evaluate Muppet. These programs are from different fields of scientific computing, utilize a variety of computational kernels, and have different levels of OpenMP parallel optimizations: some are reference implementations with the purpose of maintaining the correctness of the program, while others are manually optimized code. The programs tested include benchmark programs like Rodinia benchmarks [19], NPB-CPP [20], HPCG [21], and scientific applications such as LULESH [22], CoMD [23], CoSP2 [24], and the CLOUDSC cloud physics mini-app [25].

Among the benchmarks tested, Rodinia is a benchmark suite designed to test heterogeneous accelerators, but it also contains OpenMP version of their benchmarks for evaluation on the CPU side. In our evaluation, we choose 17 OpenMP benchmarks which run for a long enough time for performance measurement to be possible. They cover a wide range of topics, from medical imaging, fluid dynamics, data mining, to linear algebra. NPB-CPP is the C++ version of NAS Parallel Benchmarks [33] and supports various programming frameworks on shared-memory architectures including OpenMP. These benchmark programs focus on computational fluid dynamics (CFD). HPCG is a benchmark program that performs multigrid preconditioned conjugate gradient iterations, and it is a standard program to measure the performance of HPC systems. The problem sizes and original running time for these benchmark programs tested are shown in Table 1(a). We use

Table 2
Mutation speedup discovered by delta debugging (DD), Bayesian optimization (BO), and decision tree optimization (FO). Results that are not considered as having speedup are highlighted in red.

Program	Min. Time improvement			Avg. Time improvement			No. of mutations (collapse/simd/firstprivate/tile/schedule)			
	DD	BO	FO	DD	BO	FO	Original	DD	BO	FO
backprop	3.39%	3.60%	4.07%	5.09%	5.31%	5.67%	1/28/1/6/4	1/0/0/1/0	1/16/0/1/1	1/10/0/1/2
cfd	2.84%	2.84%	3.64%	2.60%	2.39%	3.20%	1/16/5/39/10	0/0/0/1/1	1/10/1/11/3	0/8/3/11/4
b+tree	0.38%	0.43%	0.47%	0.41%	0.47%	0.55%	0/30/2/6/4	0/2/0/0/0	1/13/1/0/1	0/11/2/2/2
heartwall	4.76%	4.97%	5.00%	4.70%	5.02%	5.04%	0/50/1/3/2	0/2/0/0/0	0/27/1/1/1	0/25/1/1/0
hotspot	9.34%	9.42%	9.42%	8.84%	9.53%	9.52%	0/5/0/0/0	0/3/0/0/0	0/5/0/0/0	0/4/0/0/0
hotspot3D	254.58%	257.59%	255.60%	254.64%	256.71%	255.26%	0/7/1/3/2	0/3/1/1/1	0/4/1/1/1	0/4/1/0/1
kmeans	2.13%	3.55%	1.00%	2.79%	3.03%	1.15%	0/22/0/3/2	0/16/0/0/0	0/12/0/0/0	0/10/0/1/1
lavaMD	0.58%	0.90%	0.78%	0.55%	0.83%	0.82%	0/12/1/18/2	0/6/0/3/0	0/7/1/6/1	0/5/0/3/1
leukocyte	2.63%	3.04%	3.47%	6.40%	6.37%	6.45%	0/135/3/54/4	0/9/0/0/0	0/53/1/17/2	0/67/1/16/1
lud	36.48%	38.33%	26.01%	27.80%	35.12%	22.02%	0/34/2/33/4	0/4/1/4/0	0/21/1/9/2	0/14/1/8/2
myocyte	3.67%	3.64%	4.38%	2.99%	3.10%	3.21%	0/41/2/66/2	0/40/2/22/1	0/20/0/16/0	0/24/1/19/1
nn	16.65%	17.03%	16.90%	16.52%	16.85%	16.66%	0/4/1/3/2	0/2/1/1/0	0/2/0/1/1	0/2/1/1/0
nw	0.39%	0.27%	1.56%	−0.52%	0.02%	1.72%	0/19/0/33/2	0/0/0/0/1	0/13/0/9/0	0/6/0/10/0
particlefilter	0.31%	0.15%	0.01%	1.73%	1.60%	1.51%	0/35/10/30/20	0/32/10/10/10	0/17/7/7/9	0/16/7/10/9
pathfinder	0.38%	0.40%	0.35%	2.58%	2.53%	2.38%	0/7/1/3/2	0/7/1/1/1	0/4/1/1/1	0/3/1/1/1
srad	1.74%	1.26%	1.26%	1.62%	1.62%	1.70%	2/9/2/6/4	0/1/0/0/1	0/7/0/2/2	0/6/2/2/1
streamcluster	2.30%	4.59%	3.12%	4.73%	4.85%	4.53%	0/25/2/0/0	0/2/0/0/0	0/15/0/0/0	0/13/1/0/0
BT	0.24%	−1.86%	−1.38%	0.14%	−1.84%	−1.40%	44/218/2/381/108	13/118/2/66/23	24/111/1/111/40	22/103/2/109/40
CG	1.28%	5.86%	5.02%	1.57%	2.92%	2.41%	0/18/11/27/14	0/18/9/9/7	0/9/5/9/7	0/9/8/9/5
EP	0.12%	0.11%	0.10%	0.11%	0.10%	0.10%	0/9/1/24/2	0/5/1/4/0	0/7/0/8/1	0/1/0/6/1
FT	1.80%	1.88%	1.25%	1.91%	1.88%	1.43%	1/42/5/45/12	0/6/1/1/1	0/27/3/11/3	1/23/4/10/3
LU	1.55%	2.62%	2.96%	1.43%	2.96%	2.88%	3/100/6/186/20	1/23/1/15/2	0/54/4/58/7	0/51/5/59/9
MG	15.30%	15.28%	15.67%	17.85%	17.90%	18.23%	7/66/8/39/20	3/28/4/6/3	5/34/5/12/7	4/32/6/10/9
SP	5.86%	−1.71%	−0.34%	5.89%	−1.65%	−0.81%	64/267/3/396/140	0/1/0/2/1	30/143/3/115/56	23/135/1/114/59
HPCG	4.19%	13.91%	13.21%	2.34%	8.09%	7.73%	0/63/13/81/26	0/4/0/5/0	0/36/7/25/12	0/36/7/25/12
LULESH	3.87%	2.32%	2.49%	6.29%	5.36%	5.66%	0/95/0/222/76	0/13/0/12/8	0/42/0/60/28	0/49/0/62/32
CoMD	3.34%	3.91%	4.36%	7.37%	7.86%	9.03%	0/78/13/132/30	0/24/3/10/4	0/32/8/36/12	0/47/6/36/12
CoSP2	3.80%	4.55%	6.27%	4.22%	5.10%	5.46%	0/67/9/117/22	0/17/0/12/1	0/32/5/34/10	0/44/4/36/8
CLOUDSC	1.07%	1.07%	1.20%	1.16%	3.46%	1.28%	0/58/0/111/0	0/28/0/19/0	0/34/0/34/0	0/33/0/33/0

a combination of existing and customized result verification routines to determine the correctness of the results from mutated program variants.

Among the scientific applications tested, LULESH is a proxy application simulating the Shock Hydrodynamics Challenge Problem. CoMD is a proxy application implementing classical molecular dynamics algorithms and workloads as used in materials science. CoSP2 is a reference implementation for quantum molecular dynamics (QMD) electronic structure code. Lastly, CLOUDSC is a standalone mini-app of the ECMWF cloud microphysics parameterization for its Integrated Forecasting System (IFS). Evaluating these programs may show the efficacy of Muppet in helping software developers in scientific computing optimize the parallel performance of their programs. Again, the problem sizes and original running time for these applications tested are shown in Table 1(b). As for correctness checks, for both LULESH and CoMD, we use the approach presented in [34] to determine the correctness of the program. For LULESH, we consider iteration count, final origin energy, and TotalAbsDiff as the output; for CoMD, we use the final energy as output. For CoSP2, we consider the AAN and fraction values to determine whether the mutated program variant runs correctly. And lastly, for CLOUDSC, we use its internal verification routine for this purpose.

5.1.2. Algorithm parameters

We use delta debugging, Bayesian optimization, and decision tree optimization in our evaluation. Given the fact that program running time varies across the programs being evaluated, we put a tryout limit of 100 for all programs tested across all three algorithms, instead of using a total time limit. Algorithms may finish before the tryout limit. Parameters for Bayesian optimization are `n_calls=100`, `n_initial_points=10`, and `noise=''Gaussian''`, and parameters for decision tree optimization are `n_calls=100`, `n_initial_points=10`, `acq_func=''LCB''`, `kappa=1.6`, `base_estimator=''RF''`.

5.1.3. Evaluation environment

We use a workstation computer with two 14-core Intel Xeon E5-2694v3 CPUs and 32GiB of RAM, running Ubuntu 22.04. We use Clang 16.0.6 with OpenMP 5.1 support as the compiler for both source-to-source code transformation. We use both `gcc` (Rodinia, CLOUDSC) and `clang` (others) to compile the original programs and their mutated variants, to demonstrate the adaptability of our tool to different compilers. Using OpenMP 5.1 enables us to build programs with `tiling` clauses as well.

We also ensure that performance variation is minimized between program runs. We avoid CPU context switching by limiting the programs to run on hardware threads on the second CPU by forcing the `taskset -c 14-27` command in FAROS. Hardware quiescing, as defined by [35], is also performed to reduce performance fluctuations, such as turning off both simultaneous multithreading and dynamic frequency scaling.

As for running time statistic collection, each mutated variant is run 3 times and use the minimum running time as the program running time T. As a comparison, we also record the average running time for each tryout and evaluate if there is any possible discrepancy between average and minimum running time, but this statistic is not used as the fitness function output for optimization algorithms. We use the minimum running time as the output of the fitness function because as stated in [35] it is best at rejecting noise introduced by the evaluation environment, since any running time higher than the minimum must be due to such noise. However, we still measure speedup for average running time to evaluate how performance variability affects running time across all programs.

5.2. Speedup discovered by Muppet

Even though we take various measures to reduce performance variability in our evaluation system, it is still not completely removed.

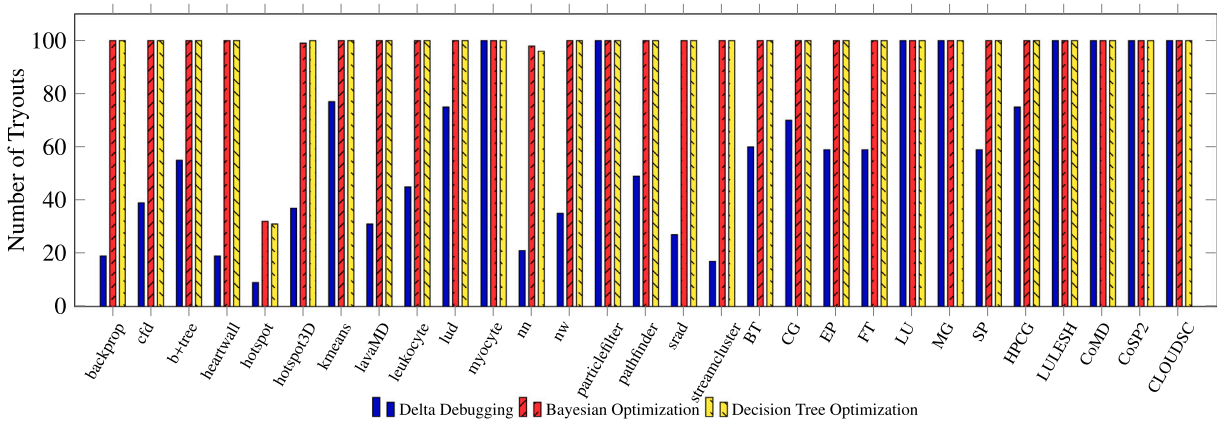


Fig. 4. The number of tryouts used by each algorithm in Muppet to terminate the optimization algorithm (maximum is 100).

Therefore, to determine if a mutated program variant shows speedup, we use the 1% threshold. If among the 3 runs, the time improvement between the minimum running time or between the average running time is less than 1%, we do not consider the current subset of mutations as speedup-inducing.

The time improvement discovered by all three algorithms with Muppet is shown in Table 2, where columns 2–4 show the time improvement when comparing the minimum running time of the best variant against the original, and columns 5–7 show the time improvement for the average running time. Our evaluation shows that there are 75.9% of programs (22 out of 29 programs; in which there are 18 out of 25 benchmark programs and 4 out of 4 scientific applications) in which delta debugging can find a subset of mutations that, when applied, can cause speedup while maintaining the correctness of the program. The other 7 programs show negligible (<1.01x) or no speedup, as shown in red background in Table 2. The largest speedup observed is from the `hotspot` benchmark in Rodinia, with a 257.59% time improvement or 3.57x speedup. Three other benchmark programs have over 1.1x speedup (`lud` and `nn` in Rodinia, and `MG` in NPB-CPP).

On the other hand, both Bayesian optimization and decision tree optimization find the same number of programs—72.4% (21 out of 29 programs) in which a subset of mutations is found to induce speedup. They also find one more program, `HPCG`, with > 1.1x speedup, compared to delta debugging. The programs with discovered speedup are the same across three algorithms, except in `SP` where delta debugging finds a subset of mutations that cause speedup, but the other two algorithms cannot.

Next, we compare the speedup discovered in individual programs by the three algorithms. Even though the speedup discovered by these algorithms looks roughly the same for a majority of programs, after comparing speedup for all programs, we find that in the 21 programs in which all algorithms find speedup in average running time, delta debugging can only find the maximum speedup among three algorithms in 2 programs, while Bayesian optimization and decision tree optimization finds the maximum speedup in 10 and 9 programs respectively. This shows that Bayesian optimization and decision tree optimization are slightly superior in finding the maximum speedup (when they can find any speedup at all). Also interesting to note, in some programs, such as `backprop` and `LULESH`, the speedup discovered for average running time is higher than the speedup discovered for minimum running time, which may suggest a reduction in performance variability in mutated program variants.

Results: Of all 29 programs tested, delta debugging can discover a subset of mutations that cause speedup in 22 (75.9%) of them; Bayesian optimization and decision tree optimization can discover speedup in 21 (69%) of them. The highest speedup discovered is 3.57x in `hotspot`. Bayesian optimization and decision tree optimization are slightly better than delta debugging in finding the highest speedups.

5.3. Time comparison between different algorithms

The time complexity of all algorithms in Muppet is defined in Section 3.4.1 but they are all subject to performance variability in the fitness function which in Muppet is the measured running time. Therefore it is preferable to look into the actual performance of these algorithms in programs tested.

We collect information on the number of total tryouts attempted for each program by each algorithm with the 100 tryout limit, and the statistics are shown in Fig. 4. For programs with a smaller amount of possible mutations, such as most benchmarks in Rodinia except `myocyte` and `particlefilter`, delta debugging can terminate before 100 tryouts. Even for NAS Parallel Benchmarks, there are still programs like `FT` and `SP` that terminate early. Meanwhile, for the other two algorithms, only 2 (`hotspot` and `nn` in Rodinia benchmarks) out of 29 programs terminated before 100 tryouts. This shows delta debugging is better than the other two algorithms in total time used when all three algorithms can find mutations that cause speedup in a program. In total, in 21 out of 29 programs, delta debugging uses fewer tryouts to finish its algorithm. In the remaining 8 programs, all algorithms terminate at the 100 tryout limit.

Our observation of logs shows that even when speedups can be discovered in relatively few tryouts for both BO and FO, the algorithms do not terminate and they continue to look for the next input that causes greater speedup. This is likely because of the noisy nature of the fitness function introduced by software and hardware environments.

Next, we evaluate how many tryouts it takes for different algorithms to first discover a subset of mutations that cause a non-negligible speedup. Fig. 5 shows the maximum speedup discovered by each algorithm for 6 programs over time. For the two programs in the top row, `hotspot3D` and `MG`, all three algorithms find a subset of mutations that cause roughly the same amount of high speedup, even though it takes decision tree optimization more tryouts to achieve that. For

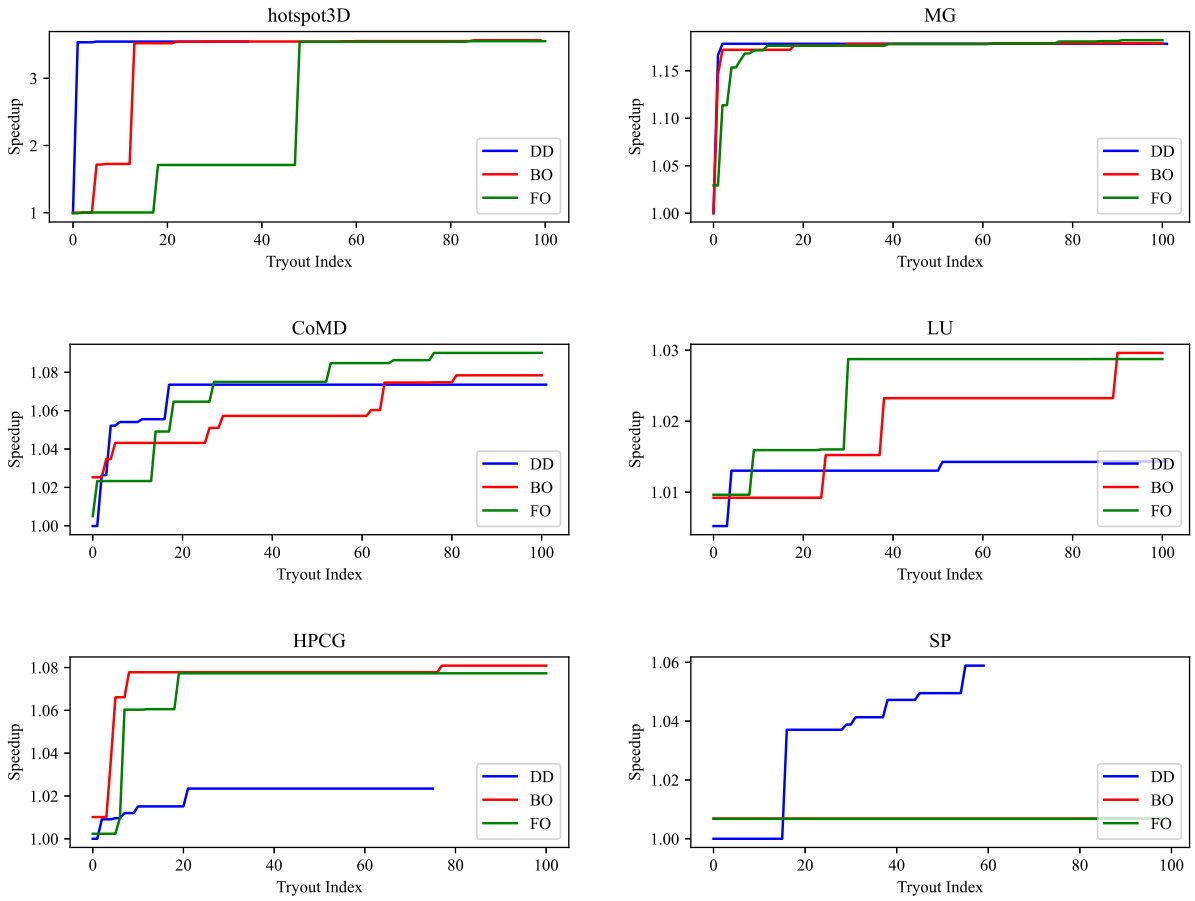


Fig. 5. Time lapse graphs showing the speedup discovered for selected programs by Muppet, using different algorithms.

the two programs in the middle row, CoMD, and LU, delta debugging uses fewer tryouts to find a subset of mutations that cause relatively high speedup, but eventually, Bayesian optimization and decision tree optimization catch up and find a subset of mutations that has an even higher speedup. For HPCG, Bayesian optimization and decision tree optimization find much higher speedup than delta debugging almost immediately and even improved their highest speedup throughout the process. The findings for these three programs are consistent with Section 5.2. Lastly, we look into an example SP where only delta debugging finds speedup. It takes delta debugging almost 20 tryouts to find the first subset that contains the speedup-inducing mutations. Before that, delta debugging can only find subsets of mutations that slow down the program.

Results: In 21 out of 29 programs, delta debugging can finish its search algorithm faster than both Bayesian optimization and decision tree optimization. For the other 8 programs, Muppet terminates at the 100 tryout limit. However, the other two algorithms, after more tryouts, can usually find the same or even higher speedup, except for SP.

5.4. Analysis of mutations found by different algorithms

Next, we look at the subset of mutations that cause the highest speedup found by each algorithm. The number of possible mutations for each program, and the number of remaining mutations in those subsets are shown in columns 8–11 in Table 2. Note that except in programs like `hotspot` or `hotspot3D` where the number of possible mutations itself is small, the number of remaining mutations in the subset of mutations found by delta debugging is always much smaller than both Bayesian optimization and decision tree optimization. Since we know from Section 5.2 that delta debugging is slightly inferior when finding a subset of mutations that cause maximum speedup, it is likely that the mutations not included in the subset found by delta debugging causes minor speedup that contributes to the maximum.

Since the number of mutations found by delta debugging is small, next we look into them for some individual programs with the highest speedups discovered to find out what kind of mutations we can find that cause significant speedup. We achieve this by looking into the runtime logs of delta debugging. In `hotspot3D`, the mutations that cause speedup consist of a firstprivate clause at the `omp parallel` directive in the `computeTempOMP` function, and some SIMD directives in the multiple-dimension loop. In `MG`, two tiling mutations, when applied to the parallelized loops in functions `psinv()` and `resid()` induce 1.03x and 1.08x speedup, while some other mutations cause minor speedup.

Lastly, we also look into SP, a program where only delta debugging finds speedup. The runtime log of both Bayesian optimization and decision tree optimization shows that a lot of subsets of mutations these algorithms generate slow down the program, sometimes the running time is longer by as much as 25%. These two algorithms would likely take significantly more than 100 tryouts to eliminate all mutations with negative speedup, while delta debugging quickly isolates the 4 out of 870 mutations that cause speedup with only 60 tryouts. These few

mutations are from SIMD and tiling mutations in the first two parallel loops in the `lhsz()` function. This shows that delta debugging is superior when there are many mutations that slow down the program.

Results: Among three algorithms, delta debugging isolates mutations causing significant speedup most quickly. Investigation into mutations that cause the highest speedups in programs evaluated shows that tiling mutations cause some of the most significant speedups discovered, while other mutations also contribute.

5.5. Discussion of results and limitations

Our evaluation shows that delta debugging is the fastest among all algorithms while discovering speedups in more programs tested. However, Bayesian optimization and decision tree optimization discover the highest speedups. Therefore, the choice between algorithms depends on the intended purpose of the end user. In a time-limited situation, it is preferable to use delta debugging to discover mutations that result in speedup; the other two algorithms are more suitable in situations where finding a speedup as high as possible is a priority.

The experimental evaluations in this paper are performed with programs that are compiled with a fixed compiler and optimization flags, both for the original program and its mutated variants. We also use fixed problem sizes and input files for all programs. Even though we use different compilers to evaluate the performance of evaluated programs and the problem sizes for our programs are varied, we cannot ensure that the mutations discovered with Muppet would also induce the same amount of speedup when either the compiler, optimization flags or runtime parameters are changed; the speedup may become larger or smaller. It is also possible that under different environments, a new performance hotspot may emerge so that running Muppet in such an environment would return different mutations to achieve speedup. Furthermore, our evaluation is performed on a workstation with a limited number of CPUs and cores. Using different hardware (CPU, RAM, etc.) or using OpenMP offload may also change the speedup and/or the mutations discovered. Nonetheless, our approach is compatible with any C/C++ compiler in any hardware and software environment, and the mutations discovered are portable across compilers.

Muppet is a dynamic tool that relies on mutation testing and it does not perform static performance modeling, nor does it perform instrumentation. The Mutation Generator, Translator, and the optimization algorithm take up negligible time during the running of Muppet, thus the running time of Muppet can be approximated as $T \cdot I$ where T is the running time of the original program and I is the number of tryouts, differed by algorithms. Thus the running time of Muppet is linearly correlated with T, and would be much longer when T is larger, such as when the program has more possible OpenMP mutations, mutations that significantly slow down the mutated variant such as those in SP described in Section 5.4, and/or when the program is run with larger problem sizes. Even with HPCG, it takes Muppet 5.5 h to complete running with Bayesian optimization (4 h with delta debugging) on the reference computer; while its original program only runs for 15.6 s. A combination of mutation testing and heuristics from program analysis may be needed to improve the search performance of Muppet.

The efficacy of Muppet also varies due to factors in program source code. Muppet is a source-level approach; this means that some mutations may already be applied manually by software developers and are no longer possible mutations as seen by Muppet. The amount of parallelism involved in the mutations may impact its efficacy as well, according to Amdahl's Law [36]. However, our experimental evaluation shows that Muppet can still assist developers in finding previously unknown optimization opportunities, even in widely used, long-established performance benchmark programs such as HPCG. Muppet can also be useful at finding optimization opportunities when migrating programs to newer OpenMP versions, as shown in examples like MG where tiling mutations that improve performance are discovered.

Lastly, due to time limitation, and given the fact that IR-level compiler optimizations are possibly already integrated into the compilers available, a comprehensive comparison between Muppet and IR-level compiler optimizations is not performed. As stated in Section 1, since Muppet is a portable, source-level approach, it can and should be used in addition to compiler optimizations. Software developers can use Muppet to discover and apply mutations into their source code to speed up their programs and avoid leaving optimization opportunities on the table, regardless of the utilized compilers and their optimizations, which may be specific to a compiler and thus not portable.

6. Related work

In the previous paper [26], we proposed an initial version of the approach of using mutation testing to optimize OpenMP program performance. In this paper, we expand on the initial version in that we introduce more mutations, one more optimization algorithm with decision tree optimization, and evaluate the efficacy of the approach in a more comprehensive methodology.

Mutation Testing. mutation testing has already been proposed to identify correctness defects [14]. The assumption in mutation testing is that a syntactic change (a mutant) can help discover programs' defects. Mutation testing, however, has not been applied deeply in HPC programs and on performance defects. Some attempts to build mutation testing for cloud systems have been reported [37]. Mutation operators (i.e., syntactic changes) have been proposed to reveal faults in small-size MPI programs [38]. With the increased use of LLVM, researchers are exploring the support of mutation testing in LLVM [39]. To the best of our knowledge, the only work that considers mutation testing for performance is [15]. However, this work does not consider parallelism and mutations in numerical (floating-point) code—these two aspects are critical to HPC applications. To the best of our knowledge, we are the first to explore using mutation testing for performance in OpenMP scientific codes.

General Auto-tuning. There are many past works on auto-tuning techniques. Typical examples include ATLAS [40], Active Harmony [41], FFTW [42], POET [43], CHILL [44], GEIST [45], OpenTuner [46], CLTune [47], Apollo [48,49], and Dutta et al. [50,51]. Their common theme is that they tune compile-time, such as tiling, or runtime parameters, such as the number of threads, presupposing a given source code representation of a program. Typical search algorithms for tuning they propose include random, grid, or Bayesian search, or various machine learning-based search models. By contrast, Muppet mutates the source code of the program, which exposes a large, combined set of both source code modifications as compile-time parameters and their possible configurations as runtime parameters to tune for. Furthermore, Muppet automates the generation of tuned source code variants without user intervention and it is the first to propose the delta debugging search algorithm for tuning. Integrating machine-learning techniques for fast searching in Muppet is an interesting future extension.

A number of papers research domain-specific tuning using code generation, alternate data layouts, or algorithmic parameters, such as [52–56] for linear algebra kernels and [57–60] for stencils. Those approaches require users to express the programs in specialized domain-specific languages amenable to tuning, which limits their generality. Muppet tunes unaltered, user-provided, general OpenMP code to generate tuning source code variants and optimizing runtime parameters.

Auto-tuning OpenMP. Specifically on OpenMP, Adaptive OpenMP [61,62], Sreenivasan et al. [63] propose OpenMP language extensions to support auto-tuning on OpenMP regions, such as scheduling policies of parallel loops, number of threads or teams. Those approaches require significant refactoring of the code and domain-specific knowledge from the programmer to successfully integrate tuning extensions and their possible configuration parameters in their OpenMP code. Instead,

MUPPET treats source code modifications as a tunable parameter and independently explores the runtime configuration space.

Bliss [64] proposes probabilistic Bayesian optimization to tune hardware (core frequency, hyperthreading) and software execution parameters (OpenMP threads, algorithmic alternatives) for the whole application, specified by the user. Bliss does not modify the program's source code and tunes all regions in unison, by contrast, MUPPET both enables source code modifications and specializes tuning to each region, since mutations are region-specific.

Scalable Record-Replay [65] is a mechanism that extracts the LLVM IR of OpenMP GPU target region kernels to tune for each kernel in parallel the GPU launch bounds as compile-time parameters, by modifying the IR to re-compile, and the number of threads/teams as runtime parameters. Performing the kind of mutations in MUPPET on LLVM IR is challenging compared to source code, which motivates our choice of a source code mutation tool. Nevertheless, the idea of extracting OpenMP regions and tuning them independently is a possible extension to MUPPET to speed up search time.

7. Conclusion

We presented MUPPET, a novel application of mutation testing aimed at improving the performance of OpenMP programs. MUPPET uses different search algorithms to apply and compose program mutations to reduce application execution time. Because program transformations are performed at the source level, MUPPET's mutations are transferable across different OpenMP implementations and compilers. We demonstrate that MUPPET is capable of identifying mutations that improve performance in 75.9% of the evaluated programs achieving a maximum speedup of 3.57x.

In the future, we plan to extend MUPPET to automatically update OpenMP code bases with the latest OpenMP features that improve performance while maintaining correctness. Currently, it is the responsibility of the code maintainer to manually update their code base to use newly available OpenMP features, which require significant manual efforts. The source code and data of MUPPET are publicly available at https://github.com/LLNL/MUPPET/.

CRediT authorship contribution statement

Dolores Miao: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Investigation, Data curation. **Ignacio Laguna:** Writing – review & editing, Supervision, Resources, Project administration, Funding acquisition. **Giorgis Georgakoudis:** Writing – original draft, Supervision, Methodology, Conceptualization. **Konstantinos Parasyris:** Writing – original draft, Supervision, Resources, Conceptualization. **Cindy Rubio-González:** Writing – review & editing, Validation, Supervision, Resources, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Dolores Miao reports financial support was provided by US Department of Energy. Ignacio Laguna reports financial support was provided by US Department of Energy. Cindy Rubio-Gonzalez reports financial support was provided by US Department of Energy. Konstantinos Parasyris reports financial support was provided by US Department of Energy. Giorgis Georgakoudis reports financial support was provided by US Department of Energy. Dolores Miao reports financial support was provided by National Science Foundation. Cindy Rubio-Gonzalez reports financial support was provided by National Science Foundation. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-863899), the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0022182, and the National Science Foundation under award CCF-2119348.

References

[1] M.A.K. Azad, N. Iqbal, F. Hassan, P. Roy, An empirical study of high performance computing (HPC) performance bugs, in: MSR, IEEE, 2023, pp. 194–206.
[2] Y. Yang, P. Xiang, M. Mantor, H. Zhou, Fixing performance bugs: An empirical study of open-source GPGPU programs, in: 2012 41st International Conference on Parallel Processing, IEEE, 2012, pp. 329–339.
[3] A. Calotoiu, T. Hoefler, M. Poke, F. Wolf, Using automated performance modeling to find scalability bugs in complex codes, in: SC, ACM, 2013, pp. 45:1–45:12.
[4] L. Adhianto, S. Banerjee, M.W. Fagan, M. Krentel, G. Marin, J.M. Mellor-Crummey, N.R. Tallent, HPCTOOLKIT: tools for performance analysis of optimized parallel programs, Concurr. Comput. Pract. Exp. 22 (6) (2010) 685–701.
[5] M. Geimer, F. Wolf, B.J. Wylie, E. Ábrahám, D. Becker, B. Mohr, The Scalasca performance toolset architecture, Concurr. Comput.: Pract. Exper. 22 (6) (2010) 702–719.
[6] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, Z.A. Matveev, Performance analysis with cache-aware roofline model in intel advisor, in: HPCS, IEEE, 2017, pp. 898–907.
[7] S.S. Shende, A.D. Malony, The TAU parallel performance system, Int. J. High Perform. Comput. Appl. 20 (2) (2006) 287–311.
[8] P.J. Mucci, S. Browne, C. Deane, G. Ho, PAPI: A portable interface to hardware performance counters, in: Proceedings of the Department of Defense HPCMP Users Group Conference, Vol. 710, Citeseer, 1999.
[9] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, S. Matsuoka, Statistical power modeling of GPU kernels using performance counters, in: International Conference on Green Computing, IEEE, 2010, pp. 115–122.
[10] A.E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, D. Lorenz, OMPT: An OpenMP tools application programming interface for performance analysis, in: International Workshop on OpenMP, Springer, 2013, pp. 171–185.
[11] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76.
[12] Y.J. Lo, S. Williams, B. Van Straalen, T.J. Ligocki, M.J. Cordery, N.J. Wright, M.W. Hall, L. Oliker, Roofline model toolkit: A practical tool for architectural and program analysis, in: International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, Springer, 2014, pp. 129–148.
[13] N. Ding, S. Williams, An instruction roofline model for GPUs, in: PMBS@SC, IEEE, 2019, pp. 7–18.
[14] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2010) 649–678.
[15] P. Delgado-Pérez, A.B. Sánchez, S. Segura, I. Medina-Bulo, Performance mutation testing, Softw. Test. Verif. Reliab. (2020) e1728.
[16] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Trans. Softw. Eng. 28 (2) (2002) 183–200.
[17] J. Mockus, Application of Bayesian approach to numerical methods of global and stochastic optimization, J. Global Optim. 4 (4) (1994) 347–365.
[18] J. Fürnkranz, Decision tree, in: Encyclopedia of Machine Learning, Springer US, Boston, MA, 2010, pp. 263–267, URL https://doi.org/10.1007/978-0-387-30164-8_204.
[19] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, K. Skadron, A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads, in: Proc. Workload Characterization (IISWC), 2010 IEEE International Symposium on, IEEE Computer Society, Atlanta, GA, USA, 2010, pp. 1–11.
[20] J. Löff, D. Griebler, G. Mencagli, G.A. de Araujo, M. Torquati, M. Danelutto, L.G. Fernandes, The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures, Future Gener. Comput. Syst. 125 (2021) 743–757.
[21] J.J. Dongarra, M.A. Heroux, P. Luszczek, High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems, Int. J. High Perform. Comput. Appl. 30 (1) (2016) 3–10.
[22] I. Karlin, A. Bhatele, J. Keasler, B.L. Chamberlain, J.D. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D.F. Richards, M. Schulz, C.H. Still, Exploring traditional and emerging parallel programming models using a proxy application, in: IPDPS, IEEE Computer Society, 2013, pp. 919–932.
[23] The Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx), CoMD - Classical molecular dynamics proxy application, 2013, https://github.com/ECP-copa/CoMD.

[24] ExMatEx, CoSP2 proxy application https://proxyapps.exascaleproject.org/app/cosp2/.
[25] ECMWF, Standalone mini-app of the ECMWF cloud microphysics parameterization, 2022, URL https://github.com/ecmwf-ifs/dwarf-p-cloudsc.
[26] D. Miao, I. Laguna, G. Georgakoudis, K. Parasyris, C. Rubio-González, MUPPET: Optimizing performance in OpenMP via mutation testing, in: PMAM@PPoPP, ACM, 2024, pp. 22–31.
[27] C. Rubio-González, C. Nguyen, H.D. Nguyen, J. Demmel, W. Kahan, K. Sen, D.H. Bailey, C. Iancu, D. Hough, Precimonious: tuning assistant for floating-point precision, in: W. Gropp, S. Matsuoka (Eds.), SC, ACM, 2013, p. 27.
[28] G. Lan, J.M. Tomczak, D.M. Roijers, A.E. Eiben, Time efficiency in optimization with a bayesian-evolutionary algorithm, Swarm Evol. Comput. 69 (2022) 100970.
[29] L. Breiman, Random forests, Mach. Learn. 45 (2001) 5–32.
[30] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, I. Shcherbatyi, Scikit-optimize/scikit-optimize, 2021.
[31] D. Quinlan, C. Liao, The ROSE source-to-source compiler infrastructure, in: Cetus Users and Compiler Infrastructure Workshop, in Conjunction with PACT, Vol. 2011, Citeseer, 2011, p. 1.
[32] G. Georgakoudis, J. Doerfert, I. Laguna, T.R.W. Scogland, FAROS: A framework to analyze OpenMP compilation through benchmarking and compiler optimization analysis, in: IWOMP, in: Lecture Notes in Computer Science, vol. 12295, Springer, 2020, pp. 3–17.
[33] D.H. Bailey, NAS parallel benchmarks, in: Encyclopedia of Parallel Computing, Springer, 2011, pp. 1254–1259.
[34] I. Laguna, P.C. Wood, R. Singh, S. Bagchi, GPUMixer: Performance-driven floating-point tuning for GPU scientific applications, in: ISC, in: Lecture Notes in Computer Science, vol. 11501, Springer, 2019, pp. 227–246.
[35] Performance engineering of software systems, 2018, URL https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/.
[36] D.P. Rodgers, Improvements in multiprocessor system design, ACM SIGARCH Comput. Archit. News 13 (3) (1985) 225–231.
[37] P.C. Cañizares, A. Núñez, M.G. Merayo, Mutomvo: Mutation testing framework for simulated cloud and HPC environments, J. Syst. Softw. 143 (2018) 187–207.
[38] R.A. Silva, S.d.R.S. de Souza, P.S.L. de Souza, Mutation operators for concurrent programs in MPI, in: 2012 13th Latin American Test Workshop, LATW, IEEE, 2012, pp. 1–6.
[39] A. Denisov, S. Pankevich, Mull it over: Mutation testing based on LLVM, in: ICST Workshops, IEEE Computer Society, 2018, pp. 25–31.
[40] R.C. Whaley, J.J. Dongarra, Automatically tuned linear algebra software, in: SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, IEEE, 1998, p. 38.
[41] C. Tapus, I.-H. Chung, J.K. Hollingsworth, Active harmony: Towards automated performance tuning, in: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, SC '02, IEEE Computer Society Press, Washington, DC, USA, 2002, pp. 1–11.
[42] M. Frigo, S.G. Johnson, The design and implementation of FFTW3, Proc. IEEE 93 (2) (2005) 216–231.
[43] Q. Yi, K. Seymour, H. You, R. Vuduc, D. Quinlan, POET: Parameterized optimizations for empirical tuning, in: 2007 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2007, pp. 1–8.
[44] C. Chen, J. Chame, M. Hall, CHiLL: A Framework for Composing High-Level Loop Transformations, Tech. Rep., Citeseer, 2008.
[45] J.J. Thiagarajan, N. Jain, R. Anirudh, A. Giménez, R. Sridhar, A. Marathe, T. Wang, M. Emani, A. Bhatele, T. Gamblin, Bootstrapping parameter space exploration for fast tuning, in: ICS, ACM, 2018, pp. 385–395.
[46] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, S.P. Amarasinghe, OpenTuner: an extensible framework for program autotuning, in: PACT, ACM, 2014, pp. 303–316.
[47] C. Nugteren, V. Codreanu, CLTune: A generic auto-tuner for OpenCL kernels, in: 2015 IEEE 9th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, IEEE, 2015, pp. 195–202.
[48] D. Beckingsale, O. Pearce, I. Laguna, T. Gamblin, Apollo: Reusable models for fast, dynamic tuning of input-dependent code, in: IPDPS, IEEE Computer Society, 2017, pp. 307–316.
[49] C. Wood, G. Georgakoudis, D. Beckingsale, D. Poliakoff, A. Giménez, K.A. Huck, A.D. Malony, T. Gamblin, Artemis: Automatic runtime tuning of parallel execution parameters using machine learning, in: ISC, in: Lecture Notes in Computer Science, vol. 12728, Springer, 2021, pp. 453–472.
[50] A. Dutta, J. Alcaraz, A. TehraniJamsaz, A. Sikora, E. César, A. Jannesari, Pattern-based autotuning of OpenMP loops using graph neural networks, in: AI4S, IEEE, 2022, pp. 26–31.
[51] A. Dutta, J. Alcaraz, A. TehraniJamsaz, E. César, A. Sikora, A. Jannesari, Performance optimization using multimodal modeling and heterogeneous GNN, in: HPDC, ACM, 2023, pp. 45–57.
[52] M. Gates, J. Kurzak, P. Luszczek, Y. Pei, J.J. Dongarra, Autotuning batch cholesky factorization in CUDA with interleaved layout of matrices, in: IPDPS Workshops, IEEE Computer Society, 2017, pp. 1408–1417.
[53] R. Nath, S. Tomov, J. Dongarra, E. Agullo, Autotuning dense linear algebra libraries on gpus and overview of the magma library, in: 6th International Workshop on Parallel Matrix Algorithms and Applications, PMAA'10, 2010.
[54] J. Kurzak, H. Anzt, M. Gates, J.J. Dongarra, Implementation and tuning of batched cholesky factorization and solve for NVIDIA GPUs, IEEE Trans. Parallel Distrib. Syst. 27 (7) (2016) 2036–2048.
[55] W.A. Abu-Sufah, A.A. Karim, Auto-tuning of sparse matrix-vector multiplication on graphics processors, in: ISC, in: Lecture Notes in Computer Science, vol. 7905, Springer, 2013, pp. 151–164.
[56] J.J. Dongarra, M. Gates, J. Kurzak, P. Luszczek, Y.M. Tsai, Autotuning numerical dense linear algebra for batched computation with GPU hardware accelerators, Proc. IEEE 106 (11) (2018) 2040–2055.
[57] P.S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, P. Sadayappan, Domain-specific optimization and generation of high-performance GPU code for stencil computations, Proc. IEEE 106 (11) (2018) 1902–1920, http://dx.doi.org/10.1109/JPROC.2018.2862896.
[58] K. Matsumura, H.R. Zohouri, M. Wahib, T. Endo, S. Matsuoka, AN5D: automated stencil framework for high-degree temporal blocking on GPUs, in: CGO, ACM, 2020, pp. 199–211.
[59] P.S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L. Pouchet, P. Sadayappan, On optimizing complex stencils on GPUs, in: IPDPS, IEEE, 2019, pp. 641–652.
[60] X. You, H. Yang, Z. Jiang, Z. Luan, D. Qian, DRStencil: Exploiting data reuse within low-order stencil on GPU, in: HPCC/DSS/SmartCity/DependSys, IEEE, 2021, pp. 63–70.
[61] C. Liao, D.J. Quinlan, R. Vuduc, T. Panas, Effective source-to-source outlining to support whole program empirical optimization, in: International Workshop on Languages and Compilers for Parallel Computing, Springer, 2009, pp. 308–322.
[62] G. Georgakoudis, K. Parasyris, C. Liao, D. Beckingsale, T. Gamblin, B. de Supinski, Machine learning-driven adaptive OpenMP for portable performance on heterogeneous systems, 2023, arXiv:2303.08873.
[63] V. Sreenivasan, R. Javali, M. Hall, P. Balaprakash, T.R. Scogland, B.R. de Supinski, A framework for enabling OpenMP autotuning, in: International Workshop on OpenMP, Springer, 2019, pp. 50–60.
[64] R.B. Roy, T. Patel, V. Gadepally, D. Tiwari, Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models, in: PLDI, ACM, 2021, pp. 1280–1295.
[65] K. Parasyris, G. Georgakoudis, E. Rangel, I. Laguna, J. Doerfert, Scalable tuning of (OpenMP) GPU applications via kernel record and replay, in: SC, ACM, 2023, pp. 28:1–28:14.