Performance Localisation

Brendan Cody-Kenny[*]
codykenny@gmail.com
Natural Computing Research & Applications, Smurfit Business School, University College Dublin, Ireland.

Michael O'Neill
m.oneill@ucd.ie
Natural Computing Research & Applications, Smurfit Business School, University College Dublin, Ireland.

Stephen Barrett
stephen.barrett@scss.tcd.ie
Distributed Systems Group, School of Computer Science & Statistics, Trinity College Dublin, Ireland

ABSTRACT

Profiling techniques highlight where performance issues manifest and provide a starting point for tracing cause back through a program. While people diagnose and understand the cause of performance to guide formulation of a performance improvement, we seek automated techniques for highlighting performance improvement opportunities to guide search algorithms.

We investigate mutation-based approaches for highlighting where a performance improvement is likely to exist. For all modification locations in a program, we make all possible modifications and analyse how often modifications reduce execution count. We compare the resulting code location rankings against rankings derived using a profiler and find that mutation analysis provides the higher accuracy in highlighting performance improvement locations in a set of benchmark problems, though at a much higher execution cost. We see both approaches as complimentary and consider how they may be used to further guide Genetic Programming in finding performance improvements.

ACM Reference Format:

Brendan Cody-Kenny, Michael O'Neill, and Stephen Barrett. 2018. Performance Localisation. In *GI'18: GI'18:IEEE/ACM 4th International Genetic Improvement Workshop , June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194810.3194815

1 INTRODUCTION

Improving program performance is frequently of secondary importance to improving functionality [8]. Performance optimization is most often attempted when a performance issue impinges on functional correctness or when developers notice a clear improvement opportunity [14]. Outside of these prominent scenarios, the implicit nature of how source code contributes to overall performance can allow potential performance opportunities to go unnoticed. Where modern development practices recommend separation of concerns, high cohesion, low coupling and reuse of code, it becomes increasingly unlikely that developers understand the performance characteristics of the APIs and libraries their code depends on [19]. To aid locating performance bugs and bottlenecks in code, static [11] and run-time profiling [20] have been developed. To understand performance issues, people start at profiled "bottleneck" locations and trace back through code [6] before devising a way to alleviate the issue. Though profiling can locate a performance issue, it may not always show where a performance *improvement* might be found. As fault localisation [7] has been used to guide automate bug fixing [22], we seek similar methods for localising performance [2] to drastically reduce the search space in automated performance improvement [9, 23].

The intuition behind the use of mutation is that there are locations in a program which are "levers on performance" and have some disproportionately large control over execution cost. We see our work as explorative within the gap between fine-grained localisation of only parameters [23], and higher level profiling of coarse-grained units of code such as functions [3].

Our hypothesis is that code locations which are particularly influential to program performance are likely good locations for finding performance improvements. To inspect this hypothesis, we consider to what extent code mutation can attribute performance to source code elements. Our research question follows as: Do code changes at the location of performance improvements produce different run-time characteristics than changes in other locations in a program?

1.1 Contributions

In this paper we inspect the use of two types of mutation:

- Deletion, which takes advantage of the hierarchical nature of source code. Source code statements (and any child statements) are deleted.
- Exhaustive, which makes all possible changes to each modification point in a program per an Abstract Syntax Tree (AST). For each code location, we can generate a set of program variants, one for each possible change at that location. In other words, we generate all possible first order mutants [10] for each modification point in a program.

We analyse the run-time characteristics of the resulting program variants for all mutations made at each location in the program. We evaluate profiling against 3 different analysis approaches in section 3 based on these mutation techniques:

[*]Work partially completed at Trinity College Dublin.

ACM ISBN 978-1-4503-5753-1/18/06…$15.00
https://doi.org/10.1145/3194810.3194815

- Under deletion (subsection 3.2), the difference in performance between the original and variant programs is attributed to all code which was deleted.
- Under exhaustive mutation (subsection 3.3), the number of times a program variant shows reduced execution cost is divided by the number of times a mutation results in a compiled program. This approach attributes performance change to finer-grained sub-statement modification points (AST nodes) in a program.
- For those code locations where no compilable programs can be produced with a single point mutation, we use deletion analysis values (subsection 3.4).

We evaluate these approaches on a set of test problems and find that

- profiling achieves the highest accuracy of all approaches on a small number of specific nodes, but does not generalise across all improvements in our problem set (Table 3),
- mutation analysis can, on average, better highlight possible performance improvements (Figure 2), and
- there is a significant trade-off between computation time required and accuracy (subsection 5.1)

2 ILLUSTRATIVE EXAMPLE

We take the scenario where a variable is initialised early in program execution and later determines how many times a loop executes. The execution cost of this variable initialisation is low as the line is only executed once, however a large amount of program execution cost can be attributed to the value of this variable when used as a condition for loop execution.

We construct a variant of BubbleSort with an additional redundant outer loop Figure 1a. A profiler will give this outer loop a very low value in terms of execution cost as can be seen in Figure 1b, therefore taking attention away from a prominent performance improvement opportunity.

The execution count in Figure 1b shows the number of times each statement is executed as a percentage. When an array of size 10 with all elements in reverse order is passed as `a`, lines 4 & 5 are both executed 200 times. The execution count for each statement will vary depending on the distribution of values within the array. An array of 10 values with a different ordering will produce a different execution count profile and can change the ranking of each statement with respect to the others. If a fully sorted array is passed then statements 5, 6 and 7 will not be executed. A reverse sorted array executes each line the maximum number of times possible and is expected to give the same ranking of statements as input size is increased. If profiling was used in this case to guide automated performance improvement [9], it would appear to decrease the chances of finding this performance improvement as effort is spent modifying other locations.

In contrast, deletion analysis shows how much of the program execution cost is attributable to the outer loop. Figure 1c shows the amount of execution cost that is saved when a statement (including any sub-statements) is removed. Execution cost savings are a percentage of the overall cost of executing the program. Note that as deletion analysis removes a statement inclusive of any sub-statements, percentages are cumulative. When statement 2 is removed the entire body of the method is removed, and so close to 100% of the execution cost is saved. Statement 2 receives a marginally larger percentage and is ranked ahead of statement 3. Deleting line 6 will result in a program which does not compile, in this case the line receives the execution cost saving from its parent statement as all nodes within a statement (and any sub-statements) are given the same value initially.

3 PERFORMANCE LOCALISATION TECHNIQUES STUDIED

In this section we further detail the localisation techniques that are compared in our evaluation [1] Though many static analysis techniques exist for detecting performance issues, we compare with profiling as it is the most prominent approach in industry and has been used to guide GP for automatic performance improvement [9].

3.1 Profiling

Our approach to profiling is relatively fine-grained to other approaches which may measure, for example, elapsed time for method execution. We measure the number of times each statement in a program is executed. For each source code statement, as defined in the Java language specification [4], we add an instrumentation statement. Each instrumentation statement consists of a function call with a program identifier and the line number for that location. When the instrumented program is run an execution count for each line is gathered.

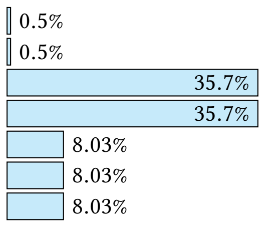
3.2 Deletion Analysis

Deletion analysis was designed in an attempt to shift focus from bottlenecks towards code which has some influence over performance. A program has a statement removed and the resulting program variant is evaluated. Deletion analysis exploits the ordered and hierarchical structure of imperative code as execution cost is attributed to statements which appear earlier in the code and to statements higher in the hierarchy. When a statement contains sub-statements, for example a "FOR", "WHILE" or "IF" statement, the inner block statement and all sub-statements are removed also. The hierarchical structure of imperative code is made accessible by using Abstract Syntax Tree (AST) parsers [21]. Statements are removed in order of their appearance in a breadth-first approach as per AST structure meaning outer loops are removed before inner loops, with the most nested code being removed last. Using deletion analysis, all AST nodes within a statement are given the same value. While hierarchical information could be similarly gathered by summing profiled execution count for nested statements, deletion mutation also may remove code which impacts performance later in the program.

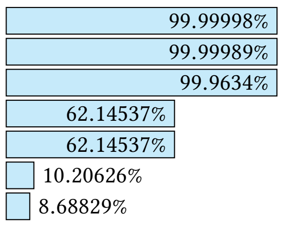
[1] Implementation available at https://github.com/codykenb/locoGP in sub-directory `locoGP_eclipse/src/locoGP/experiments` in file `ExhaustiveChange.java`

```
void sort(Integer[] a, int length){                    1
 for (int h = 0; h < 2; h++) {                         2
  for (int i=0; i < length; i++){                      3
   for (int j=0; j < length - 1; j++){                 4
    if (a[j] > a[j + 1]){                              5
     int k=a[j];                                       6
     a[j]=a[j + 1];                                    7
     a[j + 1]=k;                                       8
}}}}}                                                  9
```

(a) "BubbleLoops" problem: Bubblesort with an extra redundant outer loop



(b) Profiler: Execution frequency for each statement



(c) Deletion Analysis: Execution savings when a statement (including sub-statements) is deleted

Figure 1: BubbleLoops problem and profiles

Deletion analysis may not always be applicable for every statement in a program. Consider statement 6 in Figure 1a which initialises the variable k. Deleting this line will result in a program which does not compile and which we cannot evaluate.

3.3 Exhaustive Mutation Analysis

We apply mutation to every node in a program to produce more fine-grained rankings of AST nodes (e.g. within a statement). This yields information about statements which could not be evaluated under deletion. For example, any valid change to variable initialisation or the loop condition in the outermost loop (statement 2 in Figure 1a) is likely to show a pronounced change in the execution cost of the resulting program.

Under exhaustive mutation, a set of program variants is produced by repeatedly replacing an AST sub-tree with every other valid sub-tree as found in the original program as allowed by AST typing. For example, a statement will be replaced by all other statements, or an expression will be replaced with all other expressions in the program. A program P, is made up of a set s of all elements in the program. Let s' be a set of clones of all elements in s. For each element l in s in the program P, a variant program can be generated by exchanging l for each of the elements in s'. Alternative code elements are gleaned from the program itself but we add all language-defined [4] operators regardless of whether they are contained in the program. While this appears to produce a large number of costly program evaluations, in practice not all variant programs are compilable or evaluatable (as discussed further in subsection 5.1).

Each node in a program is attributed a value by taking the number of times a modification resulted in a program with reduced execution cost divided by the number of times a modification resulted in a compilable variant as written in Equation 1.

Example results of values attributed by exhaustive mutation analysis to statement 2 in Figure 1a `for (int h = 0; h < 2; h++) {` are shown in Table 1. It is not possible to modify some nodes in the AST as listed in the table by "-". Where no mutation can be produce a compilable program variant, the value is 0.

Table 1: Exhaustive mutation analysis example on a single line of code taken from the BubbleLoops problem

Node	Text	Value
1	`for (int h...`	1.0
2	`int h=0`	-
3	`h=0`	-
4	`h`	.6
5	`0`	.7
6	`h < 2`	-
7	`h`	.16
8	`2`	.85
9	`h++`	-
10	`h`	0

$$NodeVal = \frac{N_{executionreduction}}{N_{compiled}} \quad (1)$$

Equation 1: Exhaustive Analysis gives each node a quotient value of the number of times the execution cost is reduced, divided by the number of times a compilable program is created.

3.4 Exhaustive and Deletion Combined

Though exhaustive analysis mutates at a sub-statement level, it is still possible that no mutation at a given location is able to produce a compilable, and hence evaluatable, program variant. To alleviate this issue, we use the results of deletion analysis to "fill the gaps" in the results of exhaustive mutation analysis where no single change produced a compilable program.

4 METHODOLOGY

We use a set of benchmark problems with known performance improvements [2].

The majority of these problems were taken from online examples and improvements were found using Genetic Programming (GP)

[2]Available at: https://github.com/codykenb/locoGP in sub-directory `locoGP_eclipse/src/locoGP/problems`

[17]. AST nodes where change produces an improved program version are deemed the most important ("improvement" nodes). Improvement nodes are where GP should be applying the most mutation to have the highest chance of finding a performance improvement. We seek a technique which can find this ranking so that it can focus GP to find improvements more quickly. We compare the rankings produced from profiling and performance localisation against our idealised ranking to determine which localisation technique is most accurate. As the goal is to use mutation to highlight performance improvement locations *under further mutation*, we exclude any mutations which produce performance improvements.

4.1 Problem Set

Our problem set consists of a Huffman code-book and sort implementations[3].

Table 2 names the implementations and provides descriptive measures for each program as well as improvement types:

- **LOC** refers to the number of lines of code in the program
- **AST nodes** refers to the number of modification points in each program when it is parsed into an Abstract Syntax Tree representation [21].
- **Imp Nodes** refers to the number of nodes in a program which need to change to achieve an improved version of the program. As there are multiple changes which produce the same performance improvement, we use the improvements which give the greatest reduction in execution cost with the smallest number of modifications.
- **Improvement** refers to the largest percentage improvement in execution cost known for each program [17].
- **Improvement Types** gives a high level description of the known improvement types for each program. For loop unrolling, the important node is the containing block statement.

4.2 Test Cases

Program execution cost is affected by input size and distribution. We use a range of input sizes and distributions to ensure the profile is general. The distribution includes random, fully and reverse sorted ordering. For the Huffman code-book problem five different test cases which include repeated sequences and those without any repeated character.

4.3 Comparing Localisation Techniques

We have an idealised "best" ranking of nodes which put "improvement" nodes at the top. These top ranked improvement nodes are required to change to produce a known improvement in each program. Fractional ranking is used as all nodes in a statement jointly share a given ranking.

For each program, each localisation technique produces a ranking for all nodes. The closer an improvement node is to where it should be in the idealised ranking is used as our measure of "accuracy". The

$$PercentRankError = \frac{R_i - R_a}{N_{total}} \tag{2}$$

Equation 2: Percentile Ranking Error measure calculated for each improvement node. R_i is ideal ranking, R_a is actual derived ranking, and N_{total} is total number of nodes in the program.

distance an improvement node is from where it should be in our idealised ranking is what we use as our "ranking error" measure. We normalise the ranking error for each node by dividing it by the number of nodes in the overall program to find at what percentile the node is placed.

For each technique we compare the percentile ranking error of each important node across all problems. This gives us 48 important nodes across all problems for comparison. We also do pair-wise comparison between techniques to be sure there is a statistically significant difference between them directly. We find the difference between the approaches by subtracting the percentile rankings of the important nodes. We use a bootstrapping technique to analyse these differences. We sample randomly from these differences 100 times, with replacement, and calculate the average. This is repeated 100 times. This bootstrap approach gives an estimator for mean and approximate 95% confidence intervals are given by the 0.015 and 0.975 quantiles.

We further summarise results by looking at nodes ranked in the upper 50th percentile of all nodes. Nodes which have a ranking in the upper 50th percentile of all nodes represent instances where profiling can be said to have been "accurate". As increasing the ranking of one node reduces the ranking of another, where an improvement node is in the lower 50th percentile of all nodes then the technique can be said to be "deceived".

A normalised percentile ranking error is the distance a node is ranked from its ideal ranking, divided by the number of nodes in that program (Equation 2). This captures false negatives. We do not focus on false positives as their presence does not prevent GP from finding an improvement. What is important for GP is that all improvement nodes are ranked highly.

5 RESULTS

The four performance localisation techniques, Profiling, Deletion, Exhaustive, and Deletion with Exhaustive gap filling (Ex & De), are compared for accuracy in Table 3.

We show a split at the 50th percentile to make the point that using a probability distribution over these accuracy values will result in some cutoff point where nodes below will receive lower importance and those above will receive higher importance (in comparison to a scenario where all nodes have the same ranking or importance). We can conceive of importance being only those nodes which are in the top 1% of all nodes. In such a scenario, profiling is the only approach which would designate any improvement node as important. Profiling would be considered best in this scenario but would only highlight a single improvement node as important. The lower

[3]Long-form code listings for all programs in our problem set is available at: https://codykenb.github.io/locoGP/locoGP-ImprovementsFound.html

Table 2: Problem Improvement Overview

Problem Name	LOC	AST Nodes	Imp Nodes	Imp	Improvement Type
Insertion Sort	13	60	5	9%	Loop unrolling
Bubblesort	13	62	3	45%	Redundant Traversal (exclude sorted portion)
BubbleLoops	14	72	5	71%	Redundant Traversal (exclude sorted portion)
Selection Sort 2	16	72	4	11%	Removed redundant increments during tests
Selection Sort	18	73	4	2%	Removed redundant array access
Shell Sort	23	85	3	5%	Various changes in increment size
Radix Sort	23	100	6	3%	Reduced iteration, comparison with 0
Quick Sort	31	116	4	54%	Reduced iterations, remove tests
Cocktail Sort	30	126	4	15%	Cloned and perforated loops (loop unrolling) Redundant Traversal (exclude sorted portion)
Merge Sort	51	216	1	5%	Remove redundant array clone
Heap Sort	62	246	6	41%	Remove redundant array access and assignment
Huffman Code-book	115	411	3	43%	Redundant Traversal (exclude sorted portion)

we place the threshold for importance as a percentile, the larger the combinations of those nodes become. The more of the important nodes we want to include as important, the more program nodes we must consider. To include all important nodes we must consider all nodes in the program, which does not help us reduce the number of nodes worth considering important. The more nodes we consider, the exponentially more combinations we need to consider.

When interpreting Table 3 we consider Exhaustive with Deletion (Ex & De) to be the best as this approach places the largest number of improvement nodes in the upper 50th percentile. The three mutation-based approaches also put a majority of the improvement nodes in the upper half of all nodes.

Table 3: The accuracy of performance localisation techniques.

Accuracy	Profiler	Deletion	Exhaustive	Ex & De
99-100%	1	0	0	0
90-99%	7	8	9	11
80-90%	7	2	9	6
70-80%	3	10	7	5
60-70%	3	6	5	9
50-60%	2	4	2	5
40-50%	2	3	4	1
30-40%	6	2	5	3
20-30%	10	5	1	6
10-20%	2	2	0	2
0-10%	5	6	6	0

We further show a pair-wise comparison of the approaches using a bootstrap statistical technique (as described in subsection 4.3) over the differences of percentiles for each improvement node. Figure 2 shows pairwise differences between Profiling, Deletion, Exhaustive, and Exhaustive combined with Deletion. On average, improvement nodes are ranked roughly 2.75 percentage points higher under Deletion analysis when compared with a Profiler, 6.25 percentage points higher under Exhaustive analysis when compared with Deletion, and 3.6 percentage points higher still when using Exhaustive with Deletion.

Figure 2 also cross validates our evaluation as the differences in improvement node percentiles correlate with the ordering (though not magnitude) of which techniques are more accurate than others in Table 3. The difference between the number of improvement nodes ranked in the upper half of all nodes as shown in Table 3 (Deletion ranks more nodes in upper half than Profiling, Exhaustive more than Deletion, and Exhaustive & deletion gap filling further more still than Exhaustive alone).

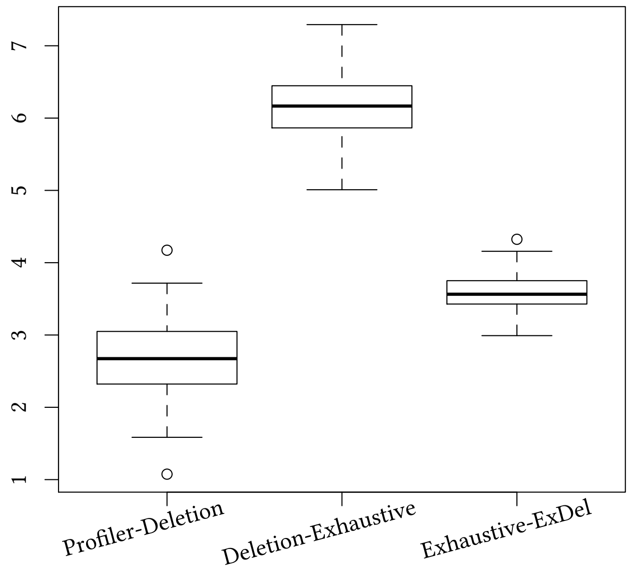


Figure 2: Comparison of the differences between node percentile rankings for the four different approaches

Table 4 shows descriptive summary values for each technique (Long form results are available [1].

Table 4: Summary

	Prof	Del	Exh	Ex & Del
Nodes: most accurate (out of 48)	**13**	10	**13**	12
Nodes: least accurate	15	19	8	**6**
Nodes: ranked in upper half	23	30	32	**36**
Nodes: ranked in lower half	25	18	16	**12**
Problems: only accurate nodes	4	4	**5**	**5**
Problems: majority deceived	6	**2**	**2**	**2**
Best on Problems	3	1	**4**	**4**

Nodes most accurate shows for how many nodes each technique is the most accurate of all techniques. Profiling has the highest accuracy values on the most nodes (13 out of a total of 48 important nodes).
Nodes least accurate sums the number of improvement nodes each technique attributes the lowest ranking of all techniques. Deletion analysis gives the lowest ranking to the most nodes when compared with all other techniques.
Nodes ranked in upper half and **Nodes ranked in lower half** show a sum of the number of nodes ranked above and below the 50th percentile respectively.
Problems with only accurate nodes counts the number of problems which do not contain any improvement nodes ranked in the lower 50th percentile.
Problems majority nodes deceived shows a less stringent count of the number of problems where a majority of the improvement nodes are ranked in the lower half of all nodes. Where a majority of nodes are given a low ranking a technique can be said to be "deceived" as to the location of an improvement.
Best on Problems refers to when a technique has given a majority of nodes the highest ranking.

Although profiling is accurate on 23 nodes across 8 problems, it is also deceived on a majority of the improvement nodes in 6 of the problems. It did however perform better than any other approach on 3 of the 12 problems including the Huffman Code-book problem which is the largest in our test set.

The use of Exhaustive analysis with deletion refinement ("Ex & Del" in table 4) was least deceived of all techniques across all nodes, with only 12 nodes lower than the 50th percentile of all nodes. It was deceived on at least 1 node in 7 of the 12 problems and was deceived on the majority of important nodes in 2 of the problems. It also has the highest accuracy on 12 of the 48 important nodes. It performs the best across 4 of the problems.

In these results we assume that if an approach is deceived on a majority of important nodes in a problem, it is likely that it will take longer to find an improvement as GP modifies other locations in the program. If half or more of the nodes are ranked highly, then it is likely that the approach will help GP find at least one of the possible improvements more quickly.

We consider a technique's ability to avoid being deceived as being more important than being the most accurate. We expect that there is some threshold value below which the use of a technique to guide a search process would lower the chances of finding an improvement. Effort spent modifying irrelevant nodes is effort that is not spent on important nodes. Due to this, we can hypothesise that a search algorithm would be delayed in finding performance improvements when focusing too much search effort on irrelevant nodes.

This is most obviously exemplified in the hand-crafted "BubbleLoops" problem, where an extra redundant outer loop has been added to Bubblesort. Profiling attributes a very low ranking to locations where simple changes which would half the execution cost of the program. Other examples which were not specifically crafted to be deceptive problems include Selection 2, Selection, Shell, Radix and Cocktail sort.

5.1 Computational Cost of Analysis

As can be seen in Figure 3, profiling is the cheapest analysis to perform as instrumentation only need be performed once and a single execution of a program is needed to gather results. Even if we use profiling to find each statement's sensitivity to input size [2], we may only need use a small number of test cases to find this information. As evaluation time dominates, we use this as our measure of computational cost. We take profiling to cost a single evaluation.

When a program is mutated, variant programs can be categorised as (1) not compilable, (2) infinite Loop, (3) run-time Error where functionality & execution cost differs from original program, (4) functionally degraded where functionality & execution cost differ. (5) more expensive (only execution cost differs from original program), (6) identical to original program in terms of functionality and execution cost and (7) less expensive in terms of execution cost. Previous results indicate that a large proportion (71 - 84%) of variant Java programs do not compile [17] although these values are found under a wider range of mutations than considered here (statement cloning is allowed).

All statements in a program can be legally deleted per the Java syntax due to the context insensitive nature of its grammar regarding statements. Deletion analysis, as implemented, requires instrumentation for every variant program. As we create a variant program by deleting each statement the number of evaluations required is almost linear to the number of statements in a program. In practice it is slightly less than linear as deleting some lines of code results in a program variant which does not compile and does not need to be evaluated. Evaluating whether a program does not compile is quicker relative to the time it takes to fully evaluate a runnable program. Deletion based localisation strikes a balance between being relatively accurate across many problems and having an execution cost linear with program size.

On the face of it, exhaustive analysis for a program containing n elements gives 2^n combinations. Evaluation is not required where a single point mutation is not possible due to the Java type system, for duplicate code elements or for programs which do not compile. Unfortunately we can not exclude programs with infinite loops[4] and runtime errors. In any case, exhaustively mutating all

[4]As we cannot determine for how long a program will execute, we somewhat arbitrarily choose a practical timeout of 2.5 times the program's execution.

code elements with all other elements in a program is practical for the relatively small programs in our test set. Although many replacements are not possible due to language typing constraints as enforced by the AST representation used, exhaustive mutation remains expensive, requiring the attempted replacement of every node with every other node. Many replacements will result in programs which can be quickly found to not compile, and therefore do not incur the comparatively large evaluation cost of repeat variant program execution with several different test input values.

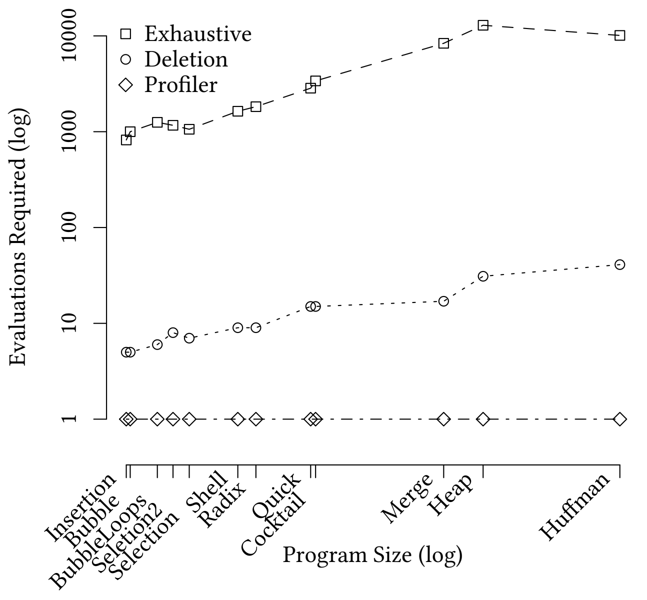


Figure 3: Comparison of the cost of analysis between Exhaustive, Deletion and Profiling. Profiling is flat, requiring only one evaluation. Deletion is linear with program size. Exhaustive is exponential in relation to the number of AST nodes free to be modified.

5.2 Threats to validity

The main threat to validity of our results is the size of the problem set with the concern being that our results do not generalise outside this set. This issue is of particular concern due to the limited variety of program type in our set; all but one of our test programs implements a sorting algorithm. Though the problem set of Sort and Huffman Codebook problems appears to be varied enough to make ranking improvement nodes highly across all problems currently unattainable, there remains a potential issue that the approach of exhaustive mutation and deletion analysis has been specialised to the algorithms in our problem set. Adding problems to the test set with particular attention paid to choosing a wider variety of problem types would reduce this concern. The length of programs is relatively small which calls into question how accuracy is affected when analysis is performed over much larger programs. As we use a sum total of execution cost it may be more difficult to measure how a mutation affects the overall cost. Our current intuition is that the accuracy of mutation-based performance localisation decreases with program length, though this would require further experimentation to validate.

The important nodes listed in our tables are sometimes part of multiple possible improvements. There are dependencies amongst some of the nodes where modifications must be made in a certain sequence to yield an improved program making some improvements easier to find than others. Not all nodes are equal, given that a change in some may produce low functionality programs and are dependent on other modifications. As not all nodes are equal in terms of dependencies a simple summation summary may not appropriately capture a localisation techniques accuracy. If a majority of important nodes in a program are highly accurately identified it may not improve search where these nodes depend on one specific node which has unfortunately been misidentified. The "importance" of nodes is thus not uniform. This concern can be addressed by a closer inspection of how “difficult” each improvement is to achieve. If an improvement requires multiple changes to the program it can be said to be more difficult to find than an improvement requiring only a single modification.

6 DISCUSSION

The major advantage of Profiling is the relatively low computational cost required. A single run of an instrumented program is enough to profile. Deletion analysis requires a program execution for each statement in a program though is more accurate on average. Exhaustive mutation is more accurate still but also considerably more expensive to perform.

The problem size we use is relatively small and there is a potential limitation especially with exhaustive mutation regarding scalability. A potential solution might be to use a hybrid of approaches. Deletion analysis could be used initially to find what statements influence execution cost the most. If removing the outermost loop reduces execution cost by some small fraction of overall cost, it may not be worth deleting and executing further nested statements. At some depth in the program subtree deletion analysis can be skipped where execution cost savings are negligible. Once deletion analysis has identified the most influential lines of code, exhaustive mutation can then be used sparingly to only distinguish between nodes within highly influential statements. Such an approach would further exploit the hierarchical structure of source code.

We use our results to say that the location of a performance bottleneck, as typically found using a profiler, does not always highlight potential performance improvements. When a performance improvement receives a low ranking, a search algorithm such as GP will be less likely to find the improvement than had there been no node ranking at all.

The cost of performing mutation can be offset in scenarios when mutation is performed for other purposes such as mutation testing [5] or genetic improvement [16, 22] (which utilises GP in many cases). Our results in this paper show that it is worth further attempting to further exploit the information generated by repeatedly executing mutated programs. Our main use case for this approach is as a guide for Genetic Programming (GP) to find performance improvements [17]. As mutation is the main driving force of the GP search process, it may be possible to localise performance during GP.

7 RELATED WORK

Our work fits between locating state with fine-grained mutation [23] and locating coarse-grained code units (functions) with profiling techniques [3] though static analysis techniques are also relevant. Using mutation to create many program variants has long been studied for software testing [5] and has recently been inspected for understanding the robustness of software [10, 18]. Closest to our approach is the use of mutation to discover "deep parameters" or locations where a modification in code relevant to program performance [23]. A deep parameter is a program mutation which affects program performance but not functionality. If program functionality changes in any measurable way, per an available test suite, the code location which was modified is removed from consideration as an interesting location for performance improvement. In contrast, our work shows that there is value in considering the location of mutations which degrade functionality but crucially also reduce execution cost. Input sensitive profiling uses progressively larger sized input values to highlight what lines of code have a particularly acute response to increased program input size [2]. A similar approach has been used with success to guide GP [9] showing the importance of performance localisation.

Static analysis is a lightweight alternative to dynamic analysis for finding performance issues. Static analysis appears to be more specific to certain types of performance issues[13, 15]. One advantage of identifying specific performance issues is that automatically fixing these performance bugs may be achieved by applying code changes which are known to frequently provide a fix [12]. In the current form of this work, when a performance bug is detected the unit of code marked as relevant to the bug is a (comparatively coarse-grained) function (or method in Java) [12]. Coarse-grained approaches which use a method or function as the smallest unit of code considered, appear more scalable for larger programs [11]. We see such approaches as complimentary where progressively less scalable approaches (such as exhaustive mutation) is only used after more scalable approaches have been used to broadly indicate what methods or libraries are associated with a performance issue.

8 CONCLUSION & FUTURE WORK

Our approach for using mutation to highlight performance improvements is general in that we do not target any specific type of code nor do we recommend any type of solution. We speculate that mutation results may be more generally applicable for different types of "performance" where mutation has a big impact on memory, network or disk usage provided these characteristics can be measured. While brute force search over all variants of a large program is unlikely to be practical, future work can hopefully allow similar performance location accuracy on far fewer program variants.

ACKNOWLEDGMENTS

This work was supported with the financial support of the Science Foundation Ireland grant 13/RC/2094.

REFERENCES

[1] Brendan Cody-Kenny and Stephen Barrett. 2016. Performance Localisation. *CoRR* abs/1603.01489v1 (2016). https://arxiv.org/abs/1603.01489v1
[2] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive profiling. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 89–98.
[3] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. 184–197.
[4] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2013. The Java Language Specification. https://docs.oracle.com/javase/specs/jls/se7/html/index.html. (Feb. 2013).
[5] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2011), 649–678.
[6] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM Press, Beijing, China, 77–88.
[7] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on Automated Software Engineering*. ACM, 273–282.
[8] Donald E Knuth. 1974. Structured Programming with go to Statements. *ACM Computing Surveys (CSUR)* 6, 4 (1974), 261–301.
[9] William B Langdon and Mark Harman. 2013. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation* (2013).
[10] William B. Langdon and Justyna Petke. 2015. Software is Not Fragile. In *Complex Systems Digital Campus E-conference, CS-DC'15 (Proceedings in Complexity)*, Paul Bourgine and Pierre Collet (Eds.). Springer, Paper ID: 356. http://cs-dc-15.org/ Invited talk, Forthcoming.
[11] Rashmi Mudduluru and Murali Krishna Ramanathan. 2016. Efficient flow profiling for detecting performance bugs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 413–424.
[12] Adrian Nistor. 2014. *Understanding, detecting, and repairing performance bugs*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
[13] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. C aramel: detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 902–912.
[14] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 237–246.
[15] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, David Grove and Steve Blackburn (Eds.). ACM, 369–378. http://dl.acm.org/citation.cfm?id=2737924
[16] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. 2014. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In *European Conference on Genetic Programming (EuroGP)*.
[17] Redacted.
[18] Eric Schulte, Zachary P Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2012. Software Mutational Robustness. *Genetic Programming and Evolvable Machines* 15, 3 (2012), 281–312.
[19] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in JavaScript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 61–72.
[20] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2015. Automating performance bottleneck detection using search-based application profiling. In *ISSTA*, Michal Young and Tao Xie (Eds.). ACM, 270–281. http://dl.acm.org/citation.cfm?id=2771783
[21] The Eclipse Foundation. 2012. Java Development Tools. http://www.eclipse.org/jdt/. (Nov. 2012).
[22] W. Weimer, T.V. Nguyen, C. Le Goues, and S. Forrest. 2009. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 364–374.
[23] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1375–1382.