Speeding up Test Execution with Increased Cache Locality

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Speeding up Test Execution with Increased Cache Locality

Panagiotis Stratis, Ajitha Rajan

School of Informatics, University of Edinburgh UK

SUMMARY

As the scale and complexity of software increases, the number of tests needed for effective validation becomes extremely large. Executing these large test suites is expensive, both in terms of time and energy. Cache misses are a significant contributing factor to execution time of software. We propose an approach that helps order test executions in a test suite in such a way that instruction cache misses are reduced. We also ensure that the approach scales to large test suite sizes.
We conduct an empirical evaluation with 20 subject programs and test suites from the SIR repository, EEMBC suite and LLVM Symbolizer, comparing execution times and cache misses with test orderings maximising instruction locality versus a traditional ordering maximising coverage and random permutations. We also assess overhead of algorithms in generating the orderings that optimise cache locality. Nature of programs and tests impact the performance gained with our approach. Performance gains were considerable for programs and test suites where the average number of different instructions executed between tests was high. We achieved an average execution speedup of 6.83% and a maximum execution speedup of 17% over subject programs with differing control flow between test executions. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

1. INTRODUCTION

-Testing to ensure the software meets its requirements is a notoriously hard and time consuming process, often representing 50% of the cost of software development [1]. As the scale and complexity of software increases, the number of tests needed for effective validation becomes extremely large, slowing down development and hindering programmer productivity with time consuming test runs. This is frequently encountered in Test-driven development (TDD) [2], a popular development practice adopted by companies like Microsoft, IBM and Google [3], [4] that requires developers to first write a test that fails before writing any new functional code. Increasing numbers of tests present a challenge for both, test generation and test execution, that make up a large fraction of the overall testing cost [5, 6]. Automating test input generation to reduce cost has received a lot of attention in the literature [7]. Reducing time and cost of test execution, on the other hand, has received little attention. Existing solutions in the literature reduce the total number of tests to be executed. This, however, can impact fault finding capability negatively [8, 9]. In this paper, we focus on **reducing test execution time** while retaining its effectiveness in fault finding.

Cache Effects on Execution Time. Applications in the present day are primarily memory speed rather than processor speed bounded, owing to rapid advances in processor performance. Cache misses are a significant consideration for memory speed [10, 11]. Powerful cache optimizations are crucial to improving the cache behavior and increasing the execution speed of these programs. Cache misses are reduced by increasing the *locality* of memory references [12], for both data and instructions. Well known optimisations include loop transformations [13, 14, 15], such as loop tiling and fusion, procedure reorderings and code layout optimisations [16, 17, 18].

Cache Locality Applied to Test Execution. Existing literature has only considered improving data/instruction locality over *single* program runs. Enhancing cache locality **across program runs** has previously not been considered and is an entirely novel contribution. We first proposed the idea of improving instruction locality across program test runs in [19]. The motivation for considering this optimisation was because of the observation that in program testing, we execute the same program several times (albeit with a different test data) increasing the chances of seeing repeated instruction sequences. We hypothesized that the knowledge of common instruction sequences between test cases can be used to help improve the performance of the instruction cache and, potentially, the entire test suite execution.

1.1. Recent work

Our approach proposed in [19] permuted tests in a test suite such that distance between neighbouring test runs is minimised. Distance is measured as number of different instructions between test runs. Permuting this way ensures that tests with similar instruction sequences will be executed in close succession, improving the chances of reusing cached instructions. We conducted a preliminary evaluation using 4 programs and associated test suites from the SIR repository [20]. We checked if the *order* in which tests are executed affects total execution time of test suite. We found that *order* of test execution *matters*, with significant time differences between worst and best permutations.

We also compared execution time of the optimised order to the distribution of execution times over 2000 random permutations of tests in the suite. Their optimised permutation outperformed a large fraction of random permutations with significant performance gains over the average and worst permutation.

1.2. Our contributions

The contributions in this paper, different from [19], are summarised as follows,

1. **Overhead and Approximation:** We conduct a detailed analysis of overhead incurred by the optimisation proposed in [19]. This was not discussed in our previous paper. We found the overhead of the existing algorithm to be significant. In this paper, we use an *approximate algorithm* to reduce overhead.
2. **Empirical Evaluation:** We compare execution times, cache misses and overhead, using programs from different benchmark families, between our original algorithm in [19] and the proposed approximation.
3. We do the following additional analyses in our empirical evaluations,

- **Compare with Branch Coverage Ordering:** We compare execution times of the locality optimised orders with a well known greedy branch coverage ordering. We also compare results of the optimised ordering with random permutations of tests.
- **Cache Misses:** We examine whether execution using locality optimised order reduces cache misses, and if the reduction in cache misses is associated with a corresponding reduction in execution time.

4. **Guideline:** Analysis of program and test suite features that are favourable for speedup gains using the proposed cache locality optimisation.

We found that the approximation algorithm proposed in this paper scales better than the original algorithm in [19] for large test suites and incurs a much smaller overhead. The execution speedup of test suite ordered with the approximation algorithm over branch coverage ordering was a maximum of 13%. We also confirmed that the execution speedups with both the approximation and original orderings were associated with a corresponding reduction in the number of cache misses. Performance gains were considerable for programs and test suites where the average distance between test runs was high.

The paper is organised as follows. Section 2 provides background on cache locality and related work. Section 3 presents the algorithms and implementations of our approach. Our experimental setup and subject programs are described in Section 4. Performance gains with respect to execution time for the different test suite orderings is presented in Section 5. Overhead of original and approximate locality ordering algorithms is analysed in Section 6. We present further analysis and a guideline to using our approach in Section 7.

2. BACKGROUND AND RELATED WORK

Present day modern applications require a vast amount of memory in order to meet their data requirements. Additionally, processor speeds have become much faster than memory speeds. As a result, execution times of many applications are memory speed, rather than processor speed bounded [21]. To help bridge the speed gap, memory systems are organised as a hierarchy with multiple layers of fast cache memory. CPU caches comprise of an *instruction cache* to speed up executable instruction fetch, and a *data cache* to speed up data fetch and store. Caches play a key role in minimizing the access latency and main memory bandwidth demand. Caches operate by retaining the most recently used data. If the processor reuses the data quickly, cache hits occur. Conversely, if it reuses the data after a long time, intervening data can evict the data from the cache, resulting in a cache miss. Cache misses cause the CPU to stall and in many applications result in a significant penalty in execution time [10, 11].

Cache misses have been shown to be inversely proportional to the locality of memory references during program execution [22]. Temporal locality is achieved by minimizing the time between references of the same memory address, i.e. the reuse of the same data within a small time frame. Spatial locality, on the other hand, is achieved when memory accesses which are close in time are also close in physical storage location. In terms of cache memory design, fetching large blocks of data (cache lines) when a cache miss occurs has the potential to increase spatial locality. However, this approach may have a negative effect on temporal locality since there is no guarantee that the additional data of the fetched cache line will be useful. Predictors and pre-fetch buffers have been proposed by the research community in a attempt to improve spatial locality without compromising temporal locality [23].

Compiler researchers have proposed the use of reuse distance as a metric to approximate cache misses [24]. Beyls et al. [25] state reuse distance of a memory access as "the number of accesses to unique addresses made since the last reference to the requested data". In a fully associative cache with n lines, a reference with reuse distance $d < n$ will hit, and with $d \geq n$ will miss. The concept of cache re-use has primarily been used in the context of data locality.

In the early 1990s, compiler optimisations were proposed to improve the cost of executing loops [26, 14]. These optimisations improve locality of data references in loops through:

- **Loop permutations** - If possible, change the sequence of loop iterations so that the iteration which enhances data reuse is placed innermost [27].
- **Loop tiling** - Iterations are reordered so that outer loop iterations are executed without waiting for the iterations of inner loops to complete execution. By this way, the distance reuse of data associated with the outer loops is decreased.
- **Loop fusion** - Multiple loops are merged under one.
- **Loop distribution** - Independent statements inside a single loop are separated into multiple, single-statement loops.
- **Variable padding** - Inter-variable padding refers to adjusting variable base addresses while intra-variable padding refers to modifying the size of data arrays. Both techniques are used heavily in compilers and have been identified to be effective in minimizing conflict misses in loops [27].

Procedure re-ordering and code layout optimisations are available in the literature for improving instruction spatial locality. Chang et al. [28] use dynamic profiling in conjunction with function inlining in an attempt to position instructions in such a way that spatial instruction locality is maximised. Chen et al. [29] propose a co-location technique for functions and basic blocks which are visited sequentially for achieving greater spatial locality.

Temporal locality of instructions has not been considered before, especially since existing optimisations are over a *single execution* of the program with little chance of repeated instruction sequences[*]. Temporal locality across mutliple executions was first proposed by us in [19]. Our approach presented in [19] is not meant to compete with the existing work on compiler or code layout optimisation. Instead, it is best if they are used **together** since we aim to improve *temporal* locality of *instructions* across several executions, while existing work improves temporal/spatial locality of *data* and *spatial* locality of *instructions* within a single execution.

Over the last decades, research community has focused on discovering ways for reducing the number of test cases. Code coverage metrics are being used by industry and academia to describe the degree to which a program is tested by a test suite [30]. These criteria are frequently encountered

[*]Unless the instructions occur within a for loop, in which case existing loop transformations help improve locality.

in regression [31] and black-box [32, 33, 34] testing where the number of test cases can become intractable for non-trivial software. Numerous optimisation techniques have been proposed which, based on coverage criteria, attempt to reduce the number or the size of the test cases. Test suite minimisation [35] refers to the systematic removal of test cases while ensuring that the test suite satisfies a set of test requirements. Test case prioritization [36] is less aggressive than test suite minimisation. Instead of removing test cases, it ranks them based on how much they contribute towards achieving a certain criteria, such as structural code coverage. Finally, test case reduction [37] examines each test case in isolation and attempts to remove redundant behaviour.

The main idea behind existing optimisation techniques is to achieve high coverage with as **few** test cases as possible. However, reducing the size or selecting a subset of tests in a test suite has been shown to also reduce its fault-finding capability [38]. Additionally, as systems become larger and more complex, the number of test cases needed for achieving acceptable levels of coverage is still very large [39]. Our approach reduces the execution time of a test suite by permuting and not removing any test cases therefore it retains the original fault-finding capability of the test suite. Our methodology can also be applied to already minimized test suites to futher improve their performance.

3. APPROACH

To maximise temporal re-use of instructions across test executions, our approach in [19] minimised the distance between tests, where the distance between two tests, T_i and T_j, is defined as:‘

$$D(T_i, T_j) = \frac{\textit{\#basic-blocks different between } T_i \textit{ and } T_j}{\textit{Total \#basic-blocks visited by all tests}} \quad (1)$$

For scalability reasons, basic blocks were used instead of instructions. Using this distance metric, we performed nearest neighbour analysis in order to produce a test permutation where the distance between subsequent tests is minimized. Algorithm 1 illustrates the optimisation approach from [19], that includes changes that we recently implemented to improve efficiency. The approach in [19] used set difference of visited basic blocks for computing the distance between tests. We changed this to use hamming distance by representing the set of visited basic blocks as binary vectors. These changes are captured in Steps 2 and 3. Steps 1 to 3 dynamically analyse test executions and computes the distance matrix. The remaining steps implement nearest neighbour analysis. The starting test is the one with the most unvisited neighbours.

This optimisation algorithm is quadratic in complexity with respect to number of tests. The main computational bottleneck is the calculation of the distance matrix which has $\frac{N^2}{2}$ complexity for a test suite with N tests. We found in our evaluation in Section 6 that the optimisation algorithm from [19] was unable to scale beyond 14K tests.

To allow the optimisation to be scalable, we implemented an approximate nearest neighbour algorithm which builds a multi-probe locality-sensitive hashing (LSH) index [40, 41] instead of calculating the full distance matrix. LSH is a technique for grouping points in high-dimensional space into buckets based on some distance measure (in our case the hamming distance). LSH uses special hash functions that map neighbouring objects in high dimensional space into approximately

```
Input: $N$ tests, $P$ program, $Thr$ defining cutoff distance between tests to be considered
    neighbours
Output: List $R$ with the permuted sequence of the $N$ tests
 1: For $1 \leq i \leq N$, run each test $T_i$ on $P$ and record the set of visited basic blocks for each $\{BT_i\}$
    as well as the set of basic blocks visited cumulatively by the test suite $\{BTS\}$.
 2: For $1 \leq i \leq N$, combine each $\{BT_i\}$ with $\{BTS\}$ in order to create a binary vector $\{BV_i\}$ of
    equal length for each test with each bit representing a basic block.
 3: $\forall i, j \in \{1, N\}$, build a $N \times N$ distance matrix of $T_i$ to $T_j$ such that $D(T_i, T_j)$ is the hamming
    distance of $\{BV_i\}$ and $\{BV_j\}$.
 4: From the distance matrix, select a starting test $T$ as the one that is not visited and has the most
    unvisited neighbours (i.e. $D(T_i, T_j) < Thr$).
 5: Set this to currentT, mark it as visited, and insert it into the end of list $R$.
 6: If currentT has no unvisited neighbours, go to Step 9.
 7: Pick the neighbour that is not visited and has the least distance from currentT.
 8: Go to step 5.
 9: If there are unvisited test runs in the distance matrix go to step 4.
10: Output $R$ as the permuted sequence of tests.
```

Algorithm 1: Algorithm from [19] for Optimised order

neighbouring objects in low dimensional space (neighbouring objects are mapped into the same hash bucket with high probability). Multi-probe LSH is a space efficient LSH that reduces the number of hash tables needed for search accuracy [40]. Our approximation is illustrated in Algorithm 2. Steps 1 and 2 are identical to the original algorithm but instead of computing the distance matrix, we construct a multi-probe LSH index. We pick a starting test at random and build an order using approximate nearest neighbour queried from LSH index until the index is empty.

```
Input: $N$ tests, $P$ program
Output: List $R$ with the permuted sequence of the $N$ tests
 1: For $1 \leq i \leq N$, run each test $T_i$ on $P$ and record the set of visited basic blocks for each $\{BT_i\}$
    as well as the set of basic blocks visited cumulatively by the test suite $\{BTS\}$.
 2: For $1 \leq i \leq N$, combine each $\{BT_i\}$ with $\{BTS\}$ in order to create a binary vector $\{BV_i\}$ of
    equal length for each test with each bit representing a basic block.
 3: Construct a multi-probe locality-sensitive hashing index $\{LSH\}$ from the set of data points
    $\{BV_i\}, i \in 1, N$.
 4: Select a random starting $T$ from the $\{LSH\}$ index.
 5: Set this to currentT, remove it from $\{LSH\}$, and insert it into the end of list $R$.
 6: If $\{LSH\}$ is empty, go to step 9.
 7: Query the $\{LSH\}$ in order to get the approximate nearest neighbour of currentT.
 8: Go to step 5.
 9: Output $R$ as the permuted sequence of tests.
```

Algorithm 2: Approximated order - Approximate nearest neighbour analysis

3.1. Implementation

We implemented our approach in C++11. Our implementation follows from Algorithms 1, 2 and is illustrated in Figure 1.

3.1.1. Test Analysis. For mapping each test to the set of its visited basic blocks we used Intel's Pin tool [42]. Pin is an instrumentation-based dynamic analysis framework which allows the

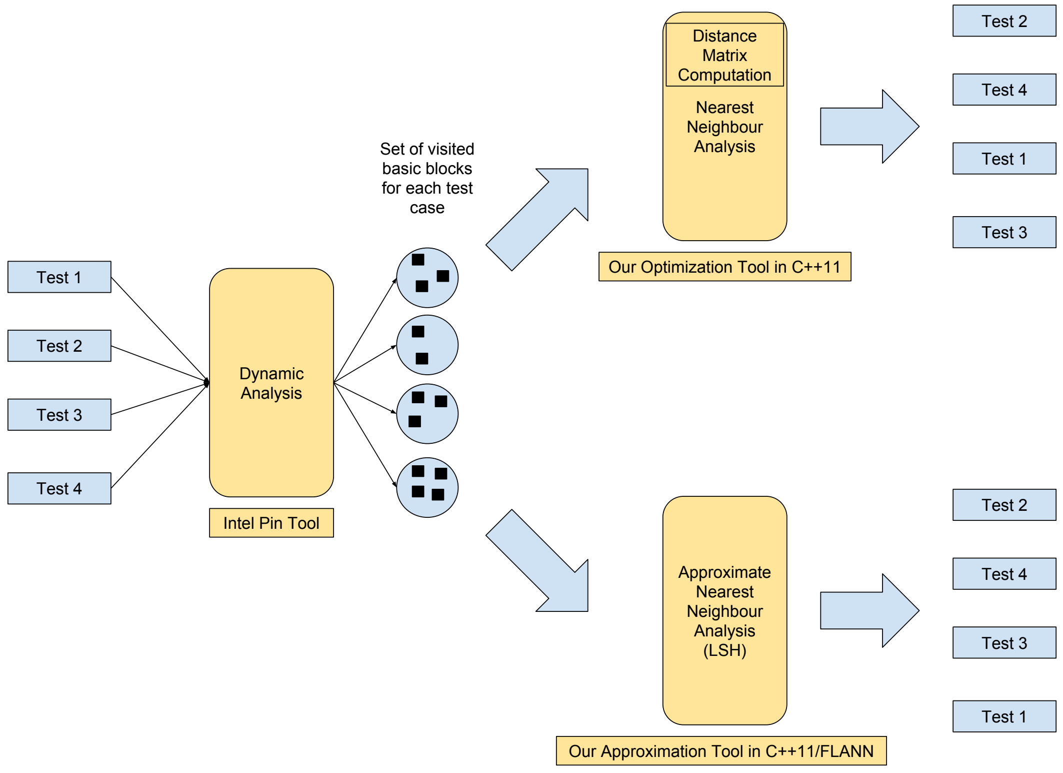
Test 1
Test 2
Test 3
Test 4
Dynamic Analysis
Intel Pin Tool
Set of visited basic blocks for each test case
Distance Matrix Computation
Nearest Neighbour Analysis
Our Optimization Tool in C++11
Test 2
Test 4
Test 1
Test 3
Approximate Nearest Neighbour Analysis (LSH)
Our Approximation Tool in C++11/FLANN
Test 2
Test 4
Test 3
Test 1

Figure 1. Implementation

development of customized dynamic program analysis tools (a.k.a Pintools). We developed a Pintool that records visited basic blocks for a program execution. Given a C/C++ program and its corresponding tests, our implementation will execute each test independently and dynamically analyse it with our Pintool.

3.1.2. Test Distance Calculation - Revised Implementation. The implementation in [19] used the standard C++ library function *std::set_symmetric_difference* for computing the distance between two tests. However, upon profiling, we found that this function does not scale adequately with respect to the size of visited basic blocks sets. We, therefore, decided to replace the set symmetric difference operation with *hamming distance* between *std::bitsets*, which is semantically equivalent in our context and has been shown to be very fast in C++ [43]. With this improved implementation using `bitsets`, the overhead of the original optimisation algorithm from [19] was significantly lower.

3.1.3. Approximate Nearest Neighbour. For locality sensitive hashing we used the C++ implementation of FLANN [44], a library for performing fast approximate nearest neighbour searches in high dimensional spaces. In our configurations, we had 12 hash tables and the length of the key in these tables was 20.

4. EXPERIMENT

We conduct our experiments over programs from different application domains to assess the following,

1. Performance - Execution time of Test Suite. We use four different types of test suites in our evaluation for performance:

- **Opt**- Test suite ordered according to the optimisation from [19], Algorithm 1.
- **Approx** - Test suite ordered using our approximation, Algorithm 2.
- **BC** - Test suite ordered greedily by an existing test adequacy measure. We use branch coverage in our evaluation since it is a widely used structural coverage metric [45]. It is worth noting that we do not reduce the test suite size based on branch coverage. We assume all tests in the suite need to be executed, so we only permute the tests based on branch coverage. The algorithm itself is a brute-force algorithm which re-orders the test cases in such a way that the highest branch coverage is achieved as quickly as possible, during the execution of the test suite. We start from the test case that covers the most branches. We then choose the next test case as the one that covers most of the remaining branches. We repeat this selection until all visited branches are covered. Any remaining tests are added at the end of the permutation in their default order.
- **Random** - We randomly permute the tests in the test suite. We generate 2000 such random permutations. This is done for programs in the SIR benchmark [20] and LLVM Symbolizer. We do not generate random permutations for programs in EEMBC benchmark [46] since the size of test suites are large, 70K tests. Execution time for large number of random permutations becomes impractical for such large test suites.

We also assess the reduction in the number of *cache misses* as a result of the optimisations.

2. Overhead of Optimisation. We assess the overhead for computing the optimised and approximated permutations with respect to increasing numbers of tests in a test suite.

4.1. Subject Programs

We assess performance and overhead over the following programs:

SIR - We use 11 programs from the SIR repository for our experiment. Programs include lexical analysers, priority schedulers, a search utility, stream text editor, a statistics program, and an aircraft collision avoidance system. Most SIR programs are accompanied by 100 to 5500 tests. Space is the only subject program in SIR with a moderately large test suite - 13585 tests. We ran the existing test suite associated with each of the SIR programs for our experiment.

EEMBC - We use the Embedded Microprocessor Benchmark Consortium (EEMBC) that provides a diverse suite of benchmarks for microprocessors, micro-controllers and embedded devices. We use 8 EEMBC benchmarks – 3 from the automotive domain (AutoBench) and 5 from the Telecommunications domain (TeleBench) of EEMBC. AutoBench is a benchmark collection for evaluating the performance of microprocessors and microcontrollers in automotive applications, and programs used in our experiment include an angle-to-time converter, a pulse-width modulator and a road speed calculator. The other 5 EEMBC benchmarks come from the telecommunications domain and comprise a convolutional encoder, a bit allocator, a viterbi decoder, a signal correlation program and a fast fourier transformer. For each of the 8 EEMBC programs, we randomly generated 70000 tests.

LLVM Symbolizer - Finally, we conducted our experiment on an LLVM tool [47], the `llvm-symbolizer` which takes as input arbitrary object files along with addresses and returns the corresponding source code locations. This tool utilizes debug info sessions and the symbol table of the input object file. We generated 432 tests which are a combination of object files from well known programs (including SIR and EEMBC) along with a set of randomly generated addresses.

Table I provides further information on the size of the different subject programs, in terms of average number of executed instructions, and the size of the test suite used.

Subject	Description	Size (Avg. Exec. Instrucs.)	#Tests	Repository
`concordance`	Utility for word indicies	3.6e+06	744	SIR
`grep`	Search utility	2.57e+06	470	SIR
`printtokens`	Lexical analyser	6.27e+03	4130	SIR
`printtokens2`	Lexical analyser	9.08e+03	4115	SIR
`replace`	Pattern matching and substitution	1.28e+04	5542	SIR
`schedule`	Priority scheduler	5.64e+03	2642	SIR
`schedule2`	Priority scheduler	1.49e+04	2710	SIR
`sed`	Stream text editor	5.36e+06	358	SIR
`space`	Interpreter for ADL	6.16e+04	13585	SIR
`tcas`	Aircraft collision avoidance system	2.23e+02	1608	SIR
`totinfo`	Statistics computation	1.89e+04	1052	SIR
`autcor00`	Cross correlation of signals	6.25e+04	70000	EEMBC
`conven00`	Convolutional encoding	5.99e+04	70000	EEMBC
`fft00`	Fast Fourier transforms	2.97e+05	70000	EEMBC
`fbital00`	Bit allocation	7.74e+04	70000	EEMBC
`viterb00`	Viterbi decoding	5.19e+05	70000	EEMBC
`a2time01`	Angle-to-time conversion	5.92e+03	70000	EEMBC
`puwmod01`	Pulse-width modulation	1.33e+06	70000	EEMBC
`rspeed01`	Road speed calculation	2.18e+07	70000	EEMBC
`llvm-symbolizer`	Address to source code conversion	1.70e+06	432	LLVM

Table I. Subject programs used in our experiment

4.2. *Measurement*

We run our experiments using a desktop computer powered by an Intel Core 2 Duo E8400 processor at 3 GHz, 32KB of Instruction Cache, and 32 KB of L1 Data Cache. The machine runs Ubuntu Server 14.04 with Linux kernel 3.16.0.33. For increased accuracy, we disable any non-critical services on the Ubuntu server while benchmarking. We measure the execution time of the algorithms and program test runs using the Unix *time* command. We report the time the under-profile program was running on the CPU (*user* statistic).

5. PERFORMANCE RESULTS

For each of the different benchmarks – SIR, EEMBC and LLVM Symbolizer, we report **performance**, using execution time, of the different test suites mentioned in Section 4 – `Opt`,

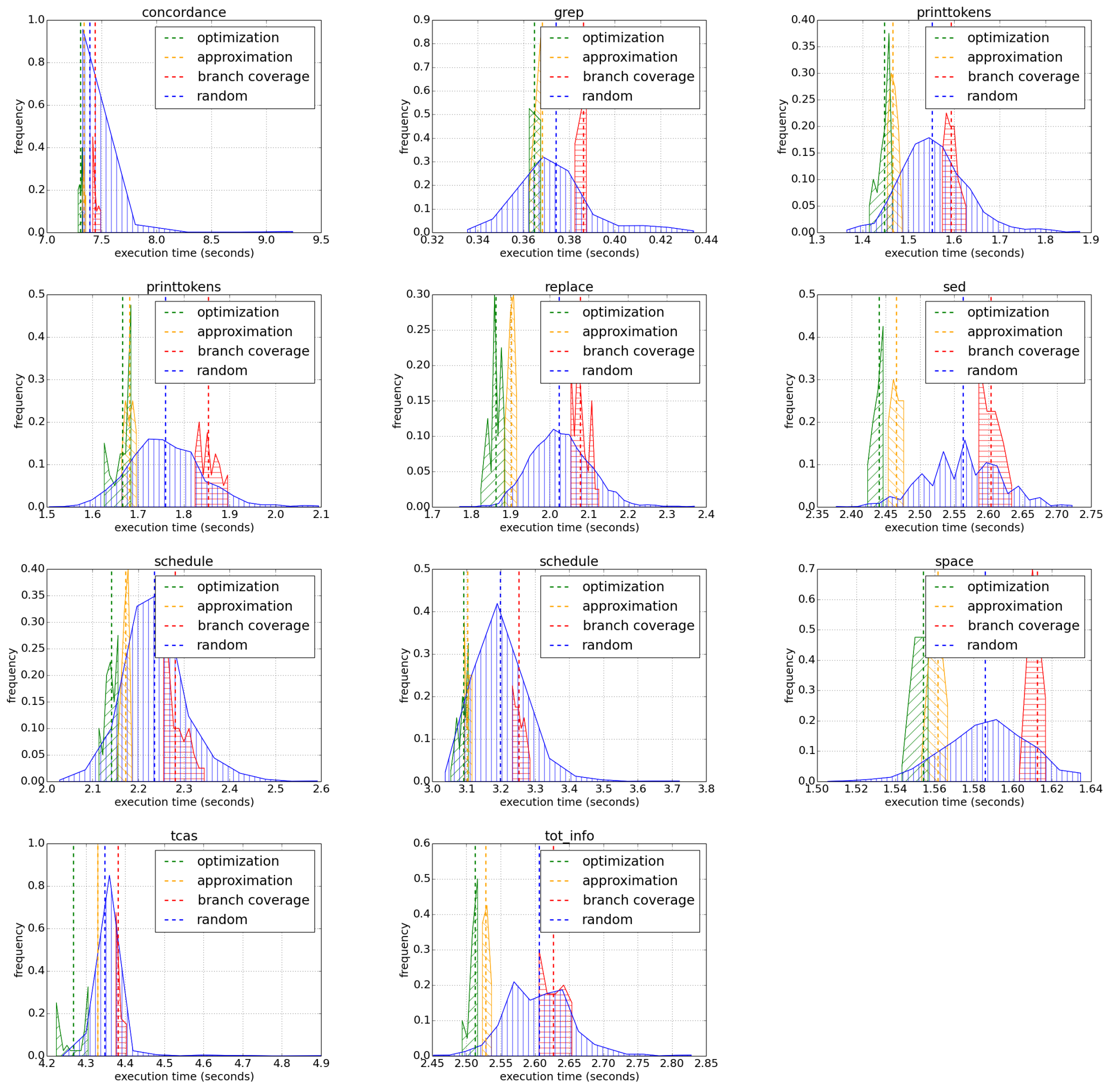

Table II. Histogram frequencies of execution time for `Opt`, `Approx`, `BC`, `Random` Test Suites for 11 SIR programs

`Approx`, and `BC`. For SIR programs and the LLVM Symbolizer, owing to smaller size of their test suites, we also report execution times of 2000 random permutations of their test suites (`Random`).

5.1. SIR

Comparison of the four different types of test suites – `Opt`, `Approx`, `BC`, and `Random`, for the 11 programs in the SIR benchmark [†] is shown in Table II. Histogram frequencies for the 2000 random permutations, and 100 runs of each of `Opt`, `Approx` and `BC` are shown. The vertical dashed line shows the median execution time over the distribution for each of the four different types of test suite. We do **not** show standard deviation, since we found that the execution times for

[†]Our earlier work [19] discussed results using only `Opt` and `Random` test suites for 4 of the 11 SIR programs shown here (`replace`, `sed`, `tcas`, `totinfo`).

all subject programs over the random test permutations were *not normally distributed*. We confirmed this by running chi-squared goodness of fit test, and the p-values for all programs were 0 (rejecting the null hypothesis that they are normally distributed). To compare independent samples of test suite execution times without assuming normal distribution, we use a non-parametric test, *Mann-Whitney-Wilcoxon test*, with a one-sided research hypothesis that helps us check whether execution time observed using one test suite type is less than the execution times observed using a different test suite type.

Observations on `Random`. It can be seen from the plots in Tables II that execution times clearly vary across random test permutations. The extent to which execution times vary is different for each program and associated random tests. Differences between the best and worst permutation execution times ranged from 9% to 29% across SIR programs. The differences observed over random permutations can be attributed to test distances being distributed over a wide range for these programs. Test suites for these programs were such that there were clusters of tests with low distances between them, i.e. they execute similar control flow paths. Distances between tests across clusters were high. As a result, random permutations that change the ordering of tests within a cluster will have little effect on the instruction locality. and those that changed the order across clusters will have a negative effect on instruction locality. The size and number of clusters will determine the magnitude of this effect.

Comparison across test suites. It is evident that for all programs, median performance of `Opt` does better than the majority of the random permutations and is very close to the best performing random permutation (left extreme of the blue curve). `Opt` executes faster than 90% of the random permutations for most SIR programs. As observed earlier, test suites for these programs have clusters of tests with low distances within, and high distances across clusters. The permutation algorithm for `Opt` ensures that tests within clusters are executed in close succession, effectively leveraging the instruction locality between them. We believe this is the primary reason for outperforming a large majority of random permutations. `Approx` also performs comparably to `Opt` on all programs, differences between their medians is between 0.5% to 2%. Among the test suite orderings, `BC` typically tends to be worst performing, achieving lesser than the median `Random` performance across all SIR programs. We believe this can be attributed to the insensitivity of the `BC` ordering to instruction similarity between tests and, as a result, incurs an increased number of instruction cache misses. We confirm this with the results from cache misses discussed in Section 5.4. For `concordance`, although median `BC` is worst performing, all 4 different test suite orderings are very close in their execution times. `Opt` ordering was only faster by 1.6% over `BC`. This is because the number of instructions executed per test for `concordance` exceeds cache size. Since multiple test executions cannot fit in the cache, it is not possible to improve instruction locality across tests with `Opt` and `Approx` orderings for this program.

Statistical Analysis As mentioned earlier, we use the *Mann-Whitney-Wilcoxon test*, with a one-sided research hypothesis to compare execution times of the different test suite orderings. We find that our results give reasonable evidence to support the following claims at 0.05 significance level,

1. `Opt` executes **faster than** {`BC`, `Approx`} for *all* 11 SIR programs.

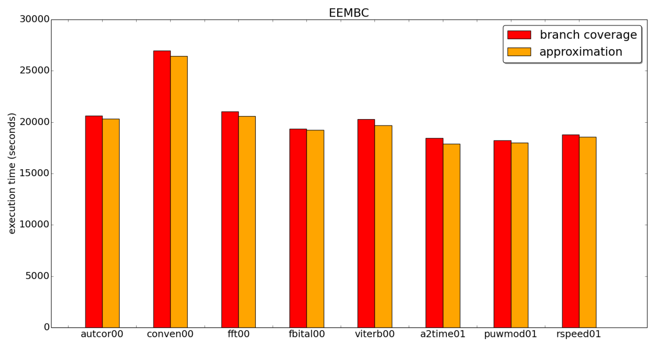

Figure 2. Comparison of execution times for `Approx`, `BC` for 8 EEMBC programs

2. `Approx` executes **faster than** `BC` for *all* 11 SIR programs.
3. `Opt` executes **faster than** `Random` on 10 SIR programs, excluding `grep`.
4. `Approx` executes **faster than** `Random` on 10 SIR programs, excluding `grep`.

`grep` did not reject the null hypotheses for claims 3 and 4 above, since there was a significant fraction of random permutations that ran slightly faster than `Opt` and `Approx`. The difference in execution times for the different test suite orderings is very small owing to low test distances observed in `grep`.

For all SIR programs, we could *not* accept the hypothesis that `Random` executions were **faster than** `BC` at 5% statistical significance. This is not surprising as a significant number of random permutations had slower executions than `BC`, as seen from the histogram frequencies in Table II.

5.2. EEMBC

Each of the subject programs in the EEMBC benchmark were accompanied by 70K randomly generated tests. Each test suite execution took more than 6.5 hours to execute. In the interest of keeping execution times practical, we did not run 2000 random permutations (`Random`) of each test suite. Additionally, we find that the overhead incurred with generating `Opt` ordering was prohibitive for test suites with 70K tests. Overhead of algorithms for `Opt` and `Approx` is discussed in Section 6. As a result, we restrict the performance discussion for EEMBC programs (with 70K tests each) to comparison of orderings generated by `Approx` and `BC`, as shown in Figure 2.

`Approx` versus `BC`. As can be seen in Figure 2, `Approx` and `BC` are comparable in performance, with `Approx` being slightly faster than `BC`. Differences in execution times are in the range of 0.50% to 3% across all 8 EEMBC programs. This may seem surprising after the results observed over SIR programs. However, on further investigation, we find that these results are to be expected. The difference in executed instructions between tests in the suite is negligible over all programs, implying similar visited basic blocks. We measured the distances among tests in the suite to confirm this and we found the average test distance (as a percentage of the total number of executed instructions by the full test suite) was in the range of 0.8% to 3% across EEMBC programs. As a result, orderings exhibit little difference since they all achieve similar locality. In comparison, average test distance for SIR programs was in the range of 8% to 39%.

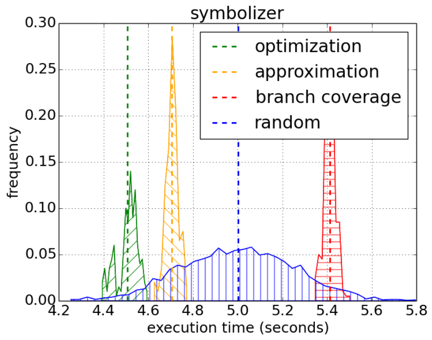

Figure 3. Histogram frequencies of execution time for `Opt`, `Approx`, `BC`, `Random` Test Suites for LLVM Symbolizer

5.3. *LLVM Symbolizer*

For LLVM Symbolizer, as with SIR, we generated all four different types of test suites – `Opt`, `Approx`, `BC`, `Random`. Figure 3 depicts the histogram frequencies for 2000 `Random` permutations and 100 runs of each of `Opt`, `Approx` and `BC`. LLVM Symbolizer showed significant execution speedup with both `Opt` (17%) and `Approx` (13%) relative to `BC`. Furthermore, `Opt` outperformed 98% of the `Random` permutations, while `Approx` outperformed 88%. Median `Opt` performance was better than median `Approx` by 4%.

Mann-Whitney-Wilcoxon test comparing test suite execution times supported the following claims with very high statistical significance, at the 0.000001 significance level (p-values were of the order of E-07 or lower),

1. `Opt` executes **faster than** {`Random`, `BC`, `Approx`}.
2. `Approx` executes **faster than** {`Random`, `BC`}.
3. `Random` executes **faster than** `BC`.

Performance gains observed over LLVM Symbolizer is highest across all benchmarks in our experiment. The superior speedup was a result of high distances between test runs in the test suite. This is further discussed in Section 7.

5.4. *Conformance with Cache Miss Rate*

The premise in locality orderings (`Opt` and `Approx`) is that they will reduce the number of instruction cache misses by increasing cache locality. This in turn will translate to faster, or reduced, execution time. We checked this premise for both `Opt` and `Approx` orderings. Cache miss rate was measured by running *Cachegrind* that is a part of Valgrind [48] on the subject programs. We find that the reduction in execution times closely follows reduction in cache misses, for the optimised orderings, relative to `BC`, over the subject programs. For instance, for `replace` in the SIR benchmark, with the `Opt` ordering, cache miss rate reduction was 18.31% compared to BC, and execution was faster by 10.14%. With the `Approx` ordering, cache miss rate reduced by 14.12% with respect to `BC`, and execution was 8.46% faster. Table III contains the speed-up and the instruction cache miss rates for all the SIR programs. For `fbital00` in EEMBC, cache miss rate reduced by 0.9% and execution time by 0.5% using the `Approx` ordering, when compared to `BC`. Unsurprisingly, cache miss rate reduction for all EEMBC programs were very small. `LLVM Symbolizer` achieved a more significant reduction in cache misses, 14.8% and 12% with `Opt` and `Approx` orderings, respectively. Corresponding speedup in execution was 17% and 13% with both orderings.

Subject	Speed-Up (%) Approx/BC	Inst. Cache Miss Rate Reduction(%) Approx/BC	Speed-Up (%) Opt/BC	Inst. Cache Miss Rate Reduction(%) Opt/BC
`concordance`	1.31	1.97	1.76	2.05
`grep`	4.66	2.63	5.57	2.65
`printtokens`	8.00	5.49	9.18	5.37
`printtokens2`	9.30	5.75	10.14	7.68
`replace`	8.46	14.12	10.39	18.31
`schedule`	4.76	2.80	6.13	2.76
`schedule2`	4.59	2.48	4.98	2.38
`sed`	5.29	3.22	6.26	3.01
`space`	3.14	1.11	3.61	1.15
`tcas`	1.17	0.50	2.58	1.1
`totinfo`	3.76	1.87	4.36	1.76

Table III. Speed-Up vs Cache Miss Rate Reduction for SIR Programs.

5.5. Impact of Data Locality

The proposed approach exploits instruction locality across tests, for the whole test suite. A natural question that may arise is, why not data locality *across* tests? Data locality optimisations involve re-using data and values; in the context of test executions, this implies re-using state and values of variables **across** test cases. Typically, after each test execution, saved state is cleaned up before starting the next test. Introducing saved state between tests introduces a previously absent data dependency that can compromise the correctness of the test suite. Safer data locality optimisations that only operate on global data, that is meant to be shared across tests, has limited potential since global data, typically, comprise a very small percentage of the total data read and written during test execution.

Although the proposed optimisations operate across tests, existing optimisations for instruction and data locality *within* each test execution are still applicable, as mentioned in Section 2 and it is best if the optimisations are used *together*. We test this hypothesis by combining our optimisation *across* test executions with an existing data locality optimisation *within* each test execution. We use the $Polly$ data locality loop optimiser [49] in LLVM for optimisations within each execution. We combined each of `Opt`, `Approx` and `BC`, with $Polly$, and measured their execution time. $Polly$ operates on the -O3 optimization level in which the compiler tries to optimize code very heavily for performance. It includes all the optimizations from the previous levels (-O1 and -O2) plus some more which involve space-speed tradeoffs. All three $Polly$-optimized permutations executed significantly faster (10% to 35%) when compared to their non-$Polly$ counterparts. However, the speedups for `Approx vs BC` and `Opt vs BC` were largely unchanged between the $Polly$ and non-$Polly$ case. The above results indicate that our approach remains useful when combined with existing data locality optimisations, that operate within a single execution. Speedup observed is more when combined with existing optimisations.

5.6. Summary

It is clear from the performance results for programs in SIR, EEMBC and LLVM Symbolizer that the order in which tests are executed **affects** execution time. The nature of programs and tests, in terms of range of distances between tests, determines the magnitude of the effect. The differences between worst and best random permutation ranged between 9% to 29% over SIR programs and 27% over LLVM symbolizer. `Opt` ordering executes fastest when compared to `Random,BC` and `Approx` orderings. However, it does not scale to large test suite sizes as discussed in Section 6. `Approx` ordering gave comparable execution times to `Opt` over all benchmarks and, in addition, is capable of scaling to large test suite sizes. Maximum speedup with `Approx`, relative to `BC`, was observed with LLVM Symbolizer (13%) that had high average distance between test runs. Least gains were observed with programs from the EEMBC benchmark that are compute-intensive with limited variation in control-flow (largely sequential). Tests, as a result, execute similar basic blocks with low distances between the different test runs. The scope for speedup with locality ordering is limited for such programs since there is little change in instruction locality achieved. Finally, we confirmed that the execution speedup of locality orderings, `Opt` and `Approx`, closely correspond to the reduction in instruction cache miss rates.

6. OVERHEAD RESULTS

In this Section, we discuss the overhead incurred in executing the algorithms for `Opt` and `Approx` orderings. For `Opt` ordering, we use the efficient implementation using *std::bitsets* mentioned in Section 3, rather than the implementation in [19]. We compare overhead (time taken) for the two orderings relative to increasing number of tests until maximum test suite size is reached. The overhead for producing `Random` permutations is negligible since it simply involves randomising the order of test ids. For `BC`, there is an overhead in monitoring branches that have been executed and then permuting the tests based on branches covered. However, as this overhead is well understood in existing literature [50], we choose to report the overhead of only our algorithms.

6.1. SIR

It is clear from Table IV that overhead of the algorithm to generate `Opt` ordering grows quadratically with the number of tests over all SIR programs. We limited our analysis to the maximum number of tests available in the repository for each program. For the `replace` program, for instance, overhead for `Opt` starts at 0.44 seconds for 554 tests and increases rapidly to 84.55 seconds for 10 times more tests. Overhead of `Approx` on the other hand, increases more slowly, only incurring 3.3 seconds for 5540 tests. This is observed uniformly over all SIR programs. Comparing the overhead of the two orderings we find overhead of `Opt` is significantly larger than that of `Approx` when number of tests is large. Considering the full test suite for all programs, `Opt` overhead is greater than `Approx` overhead by 28% (for `sed`) to 2462% (for `replace`). For subject programs, `grep` and `sed`, absolute value of `Opt` overhead is very small (< 0.25 secs) since the maximum number of tests executed with these programs is small (less than 500). At such sizes, `Opt` overhead is 28% (`sed`)

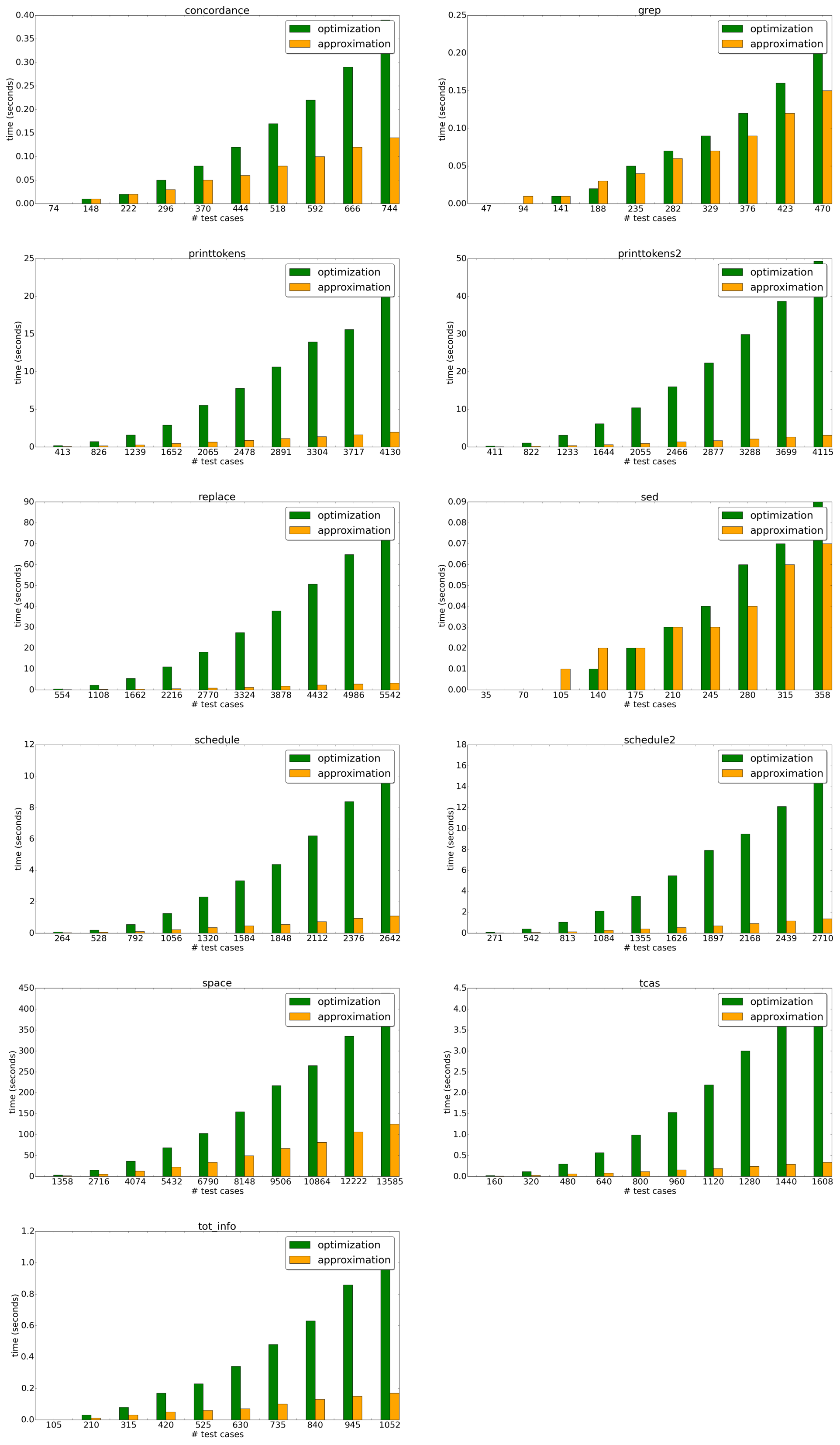

Table IV. Overhead for generating `Opt`, `Approx` orderings for increasing number of tests over 11 SIR programs

and 53% (`grep`) more than `Approx`, which is better than for other programs with larger numbers of tests.

Smaller overhead with `Approx`. Using `Approx`, rather than `Opt`, results in considerable overhead reductions across all programs. Overhead of executing the `Approx` algorithm when compared to test suite execution time is negligible for 4 of the 10 SIR programs with small test suite sizes. Overhead is 0.4 to 1.8 times test suite execution time for all other programs, except `space` where it is 7 times as much. `space` has the largest test suite with 13.5K tests, however the average executed instructions and total test suite execution time is small. As a result, the impact of `Approx` overhead is more significant. Overhead for programs and test suites with significantly longer execution times is discussed in the following Section.

6.2. EEMBC

Overhead of `Opt` and `Approx` for EEMBC benchmarks is shown in Table V for an increasing number of tests, upto 70K. We found that the `Opt` algorithm does not scale beyond 14K tests (runs out of memory). The `Approx` algorithm, on the other hand, does scale to the maximum test suite size of 70K tests, for all 8 programs. The average `Approx` overhead, across programs, as a fraction of the total execution time for 70K tests is 22.6%. Overhead of our ordering algorithm can be further reduced by running it on GPUs. We found a reduction of over 5 times in the overhead when running the `Approx` algorithm for `autcor00` with 70K tests on a NVIDIA GeForce GTX 660M with 384 CUDA cores. The overhead of the `Approx` algorithm varies significantly across the EEMBC benchmarks ranging from 3000 to 8000 seconds. We also found that the benchmarks from the automotive domain (`a2time01`, `puwmod01` and `rspeed01`) have a smaller overhead for `Approx` (around 3000 seconds) when compared to the telecommunication benchmarks (4500 to 8000 seconds). We identified the reason for this difference to be the total number of basic blocks visited **collectively** by each benchmark's test suite. In our implementation, this number defines the size of the binary vector used to represent the visited basic blocks for each test case. For the telecommunications benchmarks the total number of visited basic blocks is significantly larger than automotive benchmarks. As a result, the *approx* algorithm has to operate on a much larger volume of data which in turn leads to increased overhead for these benchmarks.

6.3. LLVM Symbolizer

The overhead incurred over LLVM Symbolizer is illustrated in Figure 4. For `Opt`, overhead ranged from 0.0004 seconds (43 tests) to 0.17 seconds (432 tests) which represents 3.7% of the full test suite execution time. For `Approx`, overhead was in the range of 0.001 seconds to 0.28 seconds (5.8% of execution time). Overhead for `Approx` is higher than `Opt` when the number of tests is small. The time taken to build LSH index in `Approx` is more significant with small test suites. It is, however, worth noting that overhead of `Approx` increases at a slower rate than `Opt` as test suite sizes get larger (owing to lower algorithmic complexity). `Opt` overhead can become prohibitive for large test suites, as observed over EEMBC programs.

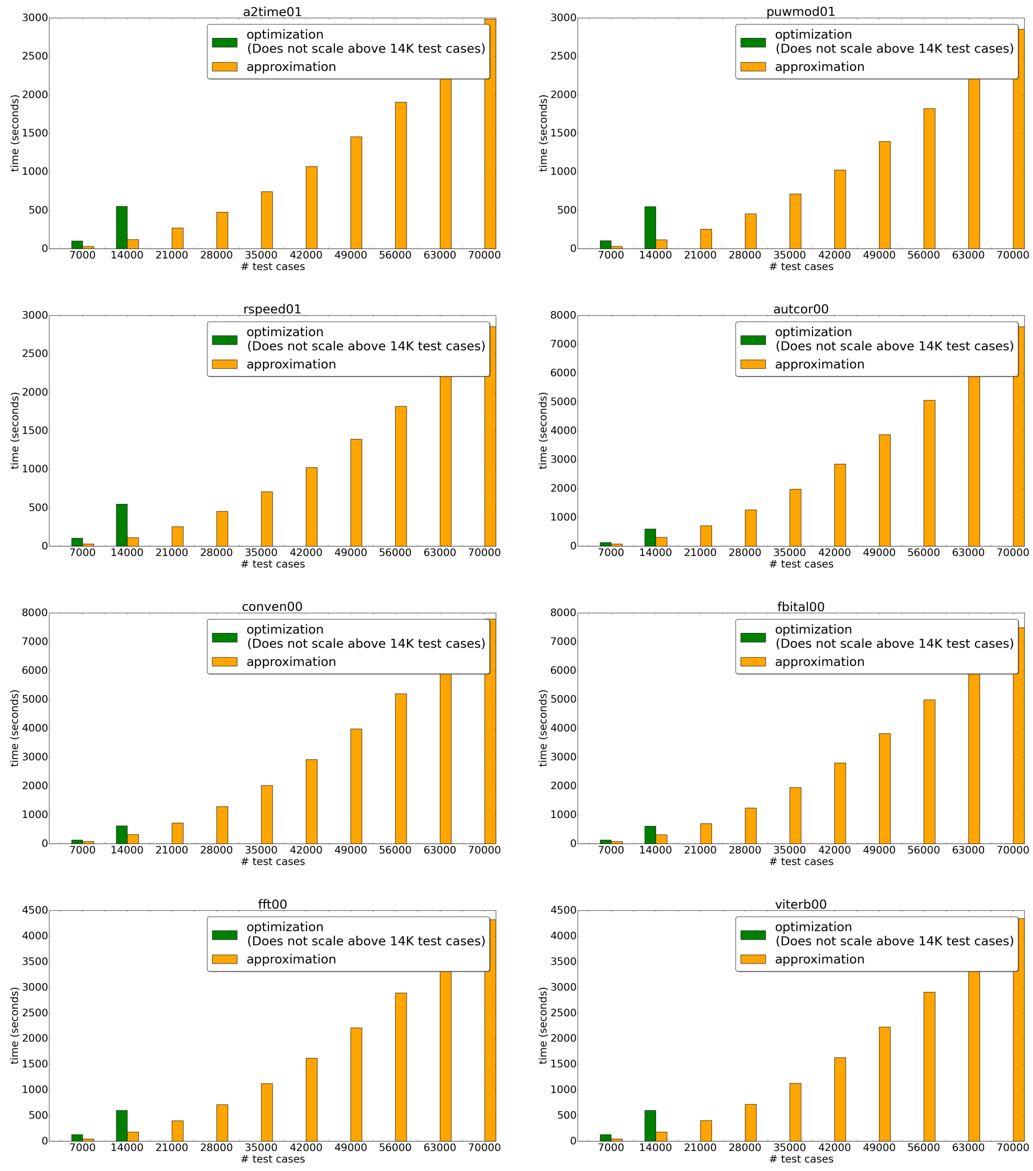

Table V. Overhead for generating `Opt`, `Approx` orderings for increasing number of tests over 8 EEMBC programs

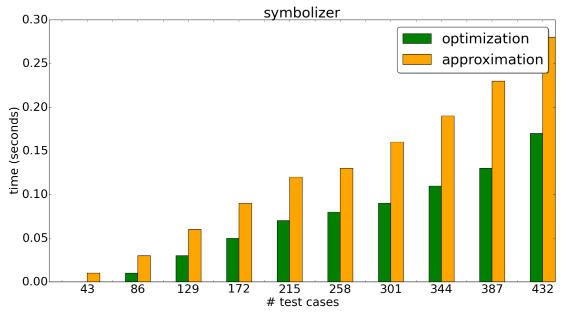

Figure 4. Overhead for generating `Opt`, `Approx` orderings for increasing number of tests for LLVM Symbolizer

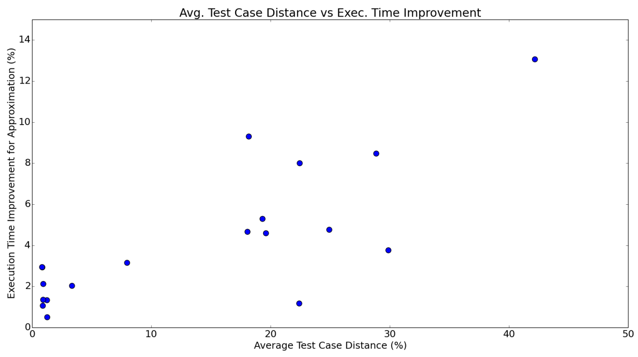

Figure 5. Test Distance versus Time Improvement for Approx ordering over BC

Overhead - Offline and Amortised. The ordering algorithms, whether it be `Approx` or `Opt`, can be performed offline (before a test suite is deployed), avoiding costly overhead during test execution phase. Additionally, the ordering, once generated, can be re-used for future test suite runs. It is common practice in embedded devices to periodically run in-situ test suites and this is further emphasized by practices like test-driven development [51].

6.4. *Summary*

Overhead of `Opt` is quadratic in size of test suite and does not scale beyond 14K tests for EEMBC programs. Overhead of `Approx` is considerably smaller and scales well to large test suites (70K tests for EEMBC). We found overhead could be further reduced with the use of GPUs. Additionally, ordering algorithms can be performed offline and overhead need not be incurred during actual test suite execution.

7. DISCUSSION

Our results in Sections 5 and 6 indicate that `Opt` and `Approx` orderings targeting cache locality result in faster execution times, than a conventional ordering like `BC`, but with varying magnitudes for the different subject programs. In this Section, we analyse reasons for this and present a metric that can be used to help predict gains and to make informed decisions on whether to apply the locality orderings.

Figure 5 shows average Test Distance (TD), and execution time improvements (speedup) of `Approx` over `BC` with full test suites for each subject program. Recall that `Opt` ordering does not scale for EEMBC programs with large test suites, so we only analyse the results for `Approx`. TD between two tests is computed as a fraction using Equation 1, number of different instructions between the two tests over total executed instructions by a test suite. For a test suite of size N, the distance matrix between tests is a square matrix (T_{ij} is symmetric to T_{ji}) with diagonal entries being 0. The number of test distances that are computed in such a matrix is $N^2/2 - N$. For each subject program, we average over all such test distances to get the `X-axis` in Figure 5.

We find that average TD is positively correlated with Approx execution time improvement, $r = 0.76$. We do not include concordance program in this computation since a single test execution exceeds cache size and our current approach is not suited for such executions. Applying our approach to test executions that exceed cache size is discussed in future work.

Average TD is a good indicator of execution speedups that can be achieved with Approx ordering when test executions fit in the cache. A higher average TD indicates the differences in instructions executed by tests in the test suite is higher. Ordering for instruction cache locality has a higher impact on performance gains for such test suites since it ensures that tests with high TD between them are not executed in succession, avoiding cache misses that result from the difference. LLVM Symbolizer has the highest average TD among subject programs of 42% and also the highest speedup with Approx of 13%. Test suites with low average TD contain tests that execute largely the same set of instructions (similar control flow). This is often seen in programs, such as those in the EEMBC suite, that are largely sequential in their control flow with only a few control flow statements. As a result, any order will have high instruction locality. Opt and Approx orderings will not result in any significant improvements for such programs and test suites. For our subject programs, we found that when average TD was low ($\leq 2\%$), execution speedup was correspondingly low ($\leq 2\%$).

Recommendations. Based on the results over our subject programs, we recommend the Approx ordering of tests in a test suite since it achieves (1) Comparable execution speedups to Opt, and (2) Scales well to large numbers of tests, as opposed to Opt. Overhead of Approx is less than that of Opt for large test suites and can be further reduced by running the algorithm on GPUs. For subject programs whose executions fit in the cache, we found average TD serves as a good guide for determining whether Approx ordering will result in reasonable performance improvements.

7.1. Threats to Validity

We see two threats to the external validity of our experiment based on the selection of programs, and choice of test suites.

We chose programs in our study that are industry standard and well-maintained like the EEMBC benchmarks, well-tested and widely-used open source programs like the LLVM Symbolizer and SIR benchmarks. Most programs in our study were also such that their execution fit in the cache. As a result, our results may only generalize to program executions satisfying this constraint. The challenge and potential solution in applying locality ordering for executions that do not fit in the cache is discussed in Section 7.2.

Another threat to external validity relates to the test suites used in our study. We used existing test suites for the SIR programs and randomly generated test suites that are controlled for test suite size for the EEMBC programs and LLVM Symbolizer. We cannot claim that the test suites we used are necessarily representative of all possible test suites. Additional research is needed to assess the performance of locality ordering with different test generation frameworks and with hand-written tests.

7.2. Future Work

The effectiveness of locality orderings, as with compiler optimisation techniques, depends on the characteristics of the program and tests. Size of the program, distances between test runs, number of tests, cache size, will all have a significant effect on the speedup achieved with our approach. For programs whose executions exceed cache size, locality ordering of tests will have little effect since mutliple executions do not fit in the cache. We discuss this challenge and a potential solution that we plan to pursue in our future work. We also discuss our approach in a parallel test execution setting.

Scaling with Size of Program. As program execution size increases, instruction locality across test runs becomes a challenge since the cache may not accommodate all the instructions from a single program run, resulting in capacity misses. To tackle this challenge, we plan to explore splitting programs into segments that fit in the cache. For instance, say we split a program P into four segments, $S1$, $S2$, $S3$, and $S4$, such that each segment fits in the cache. We run all tests on segment $S1$ storing the results, and then we run all tests on segment $S2$ using the results from $S1$ and so forth. Storing and reading intermediate results from the segment run of a test in order to execute the successor segment can be overlapped with the execution of other tests on that segment, reducing the potential bottleneck it may cause. This is a classic pipelining problem which has a well known solution with respect to instructions. We plan to suitably adapt existing ideas for instructions to segments. Running permuted tests on the segments rather than the whole program may help leverage instruction locality for large programs. We will explore the merits of this approach in our future work.

Running Tests in Parallel. It is often the case that test suites with large numbers of tests are not run sequentially on a single processor, and are, instead, launched simultaneously on multiple processors. To achieve this, the test suite is split into groups (or collections) and each group of tests is executed on a different processor. The algorithm for permuting tests, based on instruction locality, works by creating groups of tests with low distances within them. Every time we pick a new starting test (Step 4 in `Opt` and `Approx` algorithms), we start a new group. We could, therefore, easily apply our approach to create and launch groups of tests, with potentially higher instruction locality, on multiple processors. We believe the locality optimised ordering approach holds promise of time savings for executions on multiple processors; we will evaluate this hypothesis in our future work.

8. CONCLUSION

We presented an approach for ordering tests to increase cache locality across test executions. The presented approach is an approximation of ordering presented in [19]. We conducted empirical evaluations to assess execution speedups using the original approach and approximation relative to random orderings and a greedy ordering for branch coverage. We used programs from SIR, EEMBC benchmarks and an LLVM Symbolizer.

Our evaluations revealed that ordering test executions to maximise instruction locality reduces cache misses and speeds up execution. The nature of programs and tests, in terms of range of distances between test executions, determine the magnitude of the effect. The differences between worst and best random permutation execution times ranged from 9% to 29% over SIR programs and 27% for LLVM Symbolizer, providing evidence that order *matters* for test executions. Among the different orderings, `Opt` executed fastest but could not scale beyond 14K tests for EEMBC programs with 70K tests. `Approx`, on the other hand, could scale to large numbers of tests and performed comparably to `Opt`. Overhead in generating `Approx` ordering is considerably lower than the overhead for `Opt` for large test suite sizes. Execution speedup with `Approx` over `BC` was a maximum of 9% for SIR programs, 3% over EEMBC and 13% over LLVM Symbolizer. Based on our results, it is clear that increasing instruction locality with `Approx` ordering can help speedup execution of test suites. For programs whose executions fit in the cache, average test distance can be used as a guide for determining whether `Approx` ordering will result in reasonable performance gains.

REFERENCES

1. Harrold MJ. Testing: a roadmap. *Proceedings of the Conference on the Future of Software Engineering*, ACM, 2000.
2. Beck K. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
3. Nagappan N, Maximilien M, Bhat T, Williams L. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering* 2008; **13**(3).
4. Maximilien EM, Williams L. Assessing test-driven development at ibm. *Proceedings. 25th ICSE*, IEEE, 2003.
5. Young M. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.
6. Rajan A, Sharma S, Schrammel P, Kroening D. Accelerated test execution using gpus. *ACM/IEEE ASE'14*, 2014.
7. Bertolino A. Software testing research: Achievements, challenges, dreams. *Future of Software Engineering*, IEEE Computer Society, 2007; 85–103.
8. Heimdahl MPE, Devaraj G. Test-suite reduction for model based tests: Effects on test quality and implications for testing. *ASE*, 2004.
9. Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. *Proceedings of the 36th ICSE*, ACM, 2014.
10. Tiwari V, Malik S, Wolfe A, Lee MC. Instruction level power analysis and optimization of software. *VLSI Design, 1996. Proceedings., Ninth International Conference on*, 1996; 326–328, doi:10.1109/ICVD.1996.489624.
11. Vijaykrishnan N, Kandemir M, Irwin M, Kim H, Ye W. Energy-driven integrated hardware-software optimizations using simplepower. *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, 2000; 95–106.
12. Denning PJ. The locality principle. *Communications of the ACM* 2005; **48**(7):19–24.
13. Beyls K, D'Hollander E. Refactoring for data locality. *Computer* 2009; **42**(2):62–71.
14. Carr S, McKinley KS, Tseng CW. *Compiler optimizations for improving data locality*, vol. 28. ACM, 1994.
15. Grosser T, Zheng H, Aloor R, Simbürger A, Größlinger A, Pouchet LN. Polly-polyhedral optimization in llvm. *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011.
16. Ramirez A, Barroso L, Gharachorloo K, Cohn R, Larriba-Pey J, Lowney G, Valero M. Code layout optimizations for transaction processing workloads. *ACM SIGARCH Computer Architecture News*, vol. 29, ACM, 2001; 155–164.
17. Hwu WmW, Chang PP. Achieving high instruction cache performance with an optimizing compiler. *ACM SIGARCH Computer Architecture News*, vol. 17, ACM, 1989; 242–251.
18. Chen JB, Leupen BD. Improving instruction locality with just-in-time code layout. *Proceedings of the USENIX Windows NT Workshop*, 1997; 25–32.
19. Stratis P, Rajan A. Test case permutation to improve execution time. *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, IEEE, 2016; 45–50.
20. Do H, Elbaum SG, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal* 2005; **10**(4):405–435.
21. Gupta P, Lin S, McKeown N. Routing lookups in hardware at memory access speeds. *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, IEEE, 1998; 1240–1247.

22. Ghosh S, Martonosi M, Malik S. Cache miss equations: An analytical representation of cache misses. *Proceedings of the 11th international conference on Supercomputing*, ACM, 1997; 317–324.
23. Lewchuk WK. Prefetching data using profile of cache misses from earlier code executions Apr 4 2000. US Patent 6,047,363.
24. Pyo C, Lee K, Han H, Lee G. Reference distance as a metric for data locality. *High Performance Computing on the Information Superhighway, 1997. HPC Asia'97*, IEEE, 1997; 151–156.
25. Beyls K, D'Hollander E. Reuse distance as a metric for cache behavior. *Proc. of the IASTED Conf. on Parallel and Distributed Computing and Systems*, vol. 14, 2001; 350–360.
26. Wolf ME, Lam MS. A data locality optimizing algorithm. *ACM Sigplan Notices*, vol. 26, ACM, 1991; 30–44.
27. McKinley KS, Carr S, Tseng CW. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1996; **18**(4):424–453.
28. Chang PP, Mahlke SA, Chen WY, Hwu WMW. Profile-guided automatic inline expansion for c programs. *Software: Practice and Experience* 1992; **22**(5):349–369.
29. Chen Y, Patel JM. Efficient evaluation of all-nearest-neighbor queries. *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, IEEE, 2007; 1056–1065.
30. Gay G, Rajan A, Staats M, Whalen M, Heimdahl MP. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2016; **25**(3):25.
31. Rothermel G, Harrold MJ, Dedhia J. Regression test selection for c++ software. *Software Testing Verification and Reliability* 2000; **10**(2):77–109.
32. Beizer B. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.
33. Rajan A. *Coverage metrics for requirements-based testing*. University of Minnesota, 2009.
34. Rajan A. Coverage metrics to measure adequacy of black-box test suites. *21st ASE*, IEEE, 2006; 335–338.
35. Yoo S, Harman M. Pareto efficient multi-objective test case selection. *Proceedings of the 2007 international symposium on Software testing and analysis*, ACM, 2007; 140–150.
36. Elbaum S, Malishevsky AG, Rothermel G. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering* 2002; **28**(2):159–182.
37. Jones JA, Harrold MJ. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering* 2003; **29**(3):195–209.
38. Heimdahl MP, George D. Test-suite reduction for model based tests: Effects on test quality and implications for testing. *Proceedings of the 19th IEEE international conference on Automated software engineering*, 2004; 176–185.
39. Li N, Praphamontripong U, Offutt J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, IEEE, 2009; 220–229.
40. Lv Q, Josephson W, Wang Z, Charikar M, Li K. Multi-probe lsh: efficient indexing for high-dimensional similarity search. *Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment, 2007; 950–961.
41. Slaney M, Casey M. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal Processing Magazine* 2008; **25**(2):128–131.
42. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, vol. 40, ACM, 2005; 190–200.
43. Pieterse V, Kourie DG, Cleophas L, Watson BW. Performance of c++ bit-vector implementations. *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, ACM, 2010; 242–250.
44. Muja M, Lowe DG. Fast matching of binary features. *Computer and Robot Vision (CRV), 2012 Ninth Conference on*, IEEE, 2012; 404–410.
45. Baluda M, Denaro G, Pezze M. Bidirectional symbolic analysis for effective branch testing. *IEEE Transactions on Software Engineering* 2016; **42**(5):403–426.
46. Poovey J, Levy M, Gal-On S, Conte T. A benchmark characterization of the eembc benchmark suite. *Micro, IEEE* 2009; **PP**(99):1–1, doi:10.1109/MM.2009.50.
47. Lattner C, Adve V. Llvm: A compilation framework for lifelong program analysis & transformation. *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, IEEE, 2004; 75–86.
48. Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, vol. 42, ACM, 2007; 89–100.
49. Polly LLVM library. http://polly.llvm.org/index.html.

50. Tikir MM, Hollingsworth JK. Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes*, vol. 27, ACM, 2002; 86–96.
51. Ebert C, Jones C. Embedded software: Facts, figures, and future. *Computer* 2009; **42**(4).