Test Case Permutation to Improve Execution Time

Panagiotis Stratis
School of Informatics
University of Edinburgh, UK
s1329012@sms.ed.ac.uk

Ajitha Rajan
School of Informatics
University of Edinburgh, UK
arajan@staffmail.ed.ac.uk

ABSTRACT

With the growing complexity of software, the number of test cases needed for effective validation is extremely large. Executing these large test suites is expensive, both in terms of time and energy. Cache misses are known to be one of the main factors contributing to execution time of a software. Cache misses are reduced by increasing the locality of memory references. For a *single* program run, compiler optimisations help improve data locality and code layout optimisations help improve spatial locality of instructions. Nevertheless, cache locality optimisations have not been proposed and explored across several program runs, which is the case when we run several test cases.

In this paper, we propose and evaluate a novel approach to *improve instruction locality across test case runs*. Our approach measures the distance between test case runs (number of different instructions). We then permute the test cases for execution so that the distance between neighboring test cases is minimised. We hypothesize that test cases executed in this new order for improved instruction locality will reduce time consumed.

We conduct a preliminary evaluation with four subject programs and test suites from the SIR repository to answer the following questions, 1. Is execution time of a test suite affected by the order in which test cases are executed? and 2. How does time consumed in executing our permutation compare to random test case permutations? We found that the order in which test cases are executed has a definite impact on execution time. The extent of impact varies, based on program characteristics and test cases. Our approach outperformed more than 97% of random test case permutations on 3 of the 4 subject programs and did better than 93% of the random orderings on the remaining subject program. Using the optimised permutation, we saw a maximum reduction of 7.4% over average random permutation execution time and 34.7% over the worst permutation.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Cache misses, Instruction locality

1. INTRODUCTION

The number of tests needed to effectively test any non-trivial software is extremely large. With the prevalence of software in today's world and the growing complexity of systems, this number is rapidly becoming intractable. Much of the research in software testing over the last few decades has focussed on test suite[1] reduction techniques and criteria (such as coverage) that help in identifying the effective test cases to retain. This trend is particularly seen in regression testing and black-box testing where numerous optimization techniques–test case selection, test suite reduction, and test case prioritization–have been proposed to reduce testing time [23, 19, 16]. Even after using these test optimisation techniques, test suites continue to be very large and their execution is typically very time consuming. The high-level **goal** we target in this paper is,

Reduce time consumed by test suite runs.

Execution time for present day programs is memory speed rather than processor speed bounded. Missed cache hits, on both the internal and external cache, significantly slow down the execution of a program [20, 21]. Powerful cache optimizations are needed to improve the cache behavior and increase the execution speed of these programs. Based on this observation, we target the problem of reducing cache misses caused by test case runs.

Cache misses are reduced by increasing the *locality* of memory references [7], for both data and instructions. Compiler researchers proposed loop tranformation techniques such as loop tiling and fusion to improve data locality [3, 4, 1]. To enhance instruction locality, researchers proposed procedure orderings and code layout optimisations [17, 10, 5] to improve spatial locality of instructions.

Existing literature has only considered improving data/instruction locality over *single* program runs. Enhancing locality across program runs is entirely novel. We rephrase the earlier problem in terms of locality as, *improve locality of references* ***across*** *test case runs.*

In this paper, we target this locality problem in the context of testing. 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 believe the knowledge of common instruction sequences between test cases can be used to help improve the performance of the instruction cache and, potentially, the program. We propose a novel approach based on this to improve *instruction locality* across test case runs. We permute

[1]A test suite is a collection of test cases that test a program.

the test cases so that distance between neighboring test case runs is minimised. Distance is measured as number of different instructions between test runs. Our approach and algorithm for permuting test cases is discussed in Section 3.

The proposed idea for leveraging instruction cache locality in the context of test executions is entirely novel. However, the effect of test case permutations and improving instruction cache locality on execution time is not well understood or even known. We conduct a preliminary evaluation to first assess the impact of test case permutation on execution time using randomly generated permutations of a test suite. We used 4 subject programs and test workloads from the SIR repository. We also evaluated the usefulness of our approach in addressing the high-level goal by comparing execution time of our optimised permutation against average, best and worst execution times of the random permutations. Our experiments revealed that the order in which test cases are executed has an impact on execution time. The magnitude of the effect varied greatly and was dependent on program and test suite characteristics. We also found that our optimised permutation outperformed a significant proportion of random permutations in all subject programs. The reduction in execution time was in the range of 1.4% to 7.4% over average random permutation, and 13.8% to 34.7% over the worst permutation.

The rest of our paper is organised as follows. We provide background on cache misses and existing compiler technqiues to improve locality in Section 2. We describe the steps in our algorithm and our implementation in Section 3. The experiment for evaluating our approach is described in Section 4. Results and their analysis is presented in Section 5.

2. BACKGROUND AND RELATED WORK

Today's computer systems must manage a vast amount of memory to meet the data requirements of modern applications. Practically, all 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 data 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 significant penalty in execution time [20, 21].

Cache misses are reduced by increasing the *locality* of memory references. *Temporal locality* refers to memory accesses that are close in time, i.e. the reuse of a specific data within a relatively small time duration. *Spatial locality* refers to memory accesses close to each other in storage locations. Modern cache designs exploit spatial locality by fetching large blocks of data called cache lines on a cache miss. Subsequent references to words within the same cache line result in cache hits. In the following paragraphs, we present existing literature for optimising temporal and spatial locality of data references (for data cache) and instruction references (for instruction cache).

Compiler researchers have proposed the use of reuse distance as a metric to approximate cache misses [2, 15]. Beyls et al. 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.

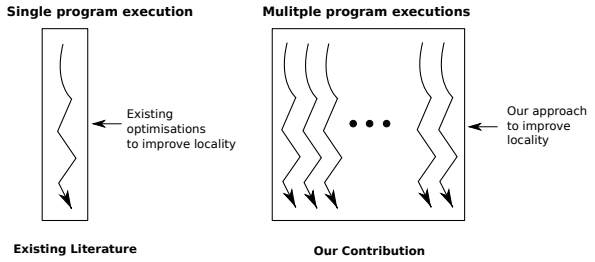


Figure 1: Existing work versus Our Contribution

In the early 1990s, compiler optimisations were proposed to improve the cost of executing loops [22, 4]. These optimisations improve locality of data references in loops through,
Loop Permutations: changing the sequence of loop iterations such that the iteration that promotes the most data reuse is positioned innermost (if it is legal).
Loop Tiling: reordering iterations such that iterations from outer loops are executed before completing all the iterations of the inner loop. Tiling reduces the number of intervening iterations and thus data fetched between data reuses [22].
Loop Fusion: takes multiple loops and combines their bodies into one loop nest.
Loop Distribution: separates independent statements in a single loop into multiple loops with single headers.
Variable padding: Inter-variable padding that adjusts variable base addresses, and intra-variable padding that modifies array dimension sizes have been proposed to eliminate conflict misses occuring in loop iteratons [18].
Data reference predictors and prefetchers have also been proposed to improve locality and hide memory latency resulting from cache misses [12, 11, 6].

To improve spatial locality of instructions, existing literature has explored procedure re-orderings and code layout optimisations. Hwu et al. use dynamic profiling, and function inlining for instruction placement that maximises sequential and spatial locality [10]. Chen et al. and Ramirez et al. achieve spatial locality by co-locating procedures or basic blocks that are activated sequentially [5, 17].

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[2]. Figure 1 illustrates the difference between existing work and our approach. Our approach targets optimisation of temporal locality of instructions across *several executions* of the program. Our approach presented in Section 3 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.

Related work in the field of software testing has proposed numerous optimization techniques–test case selection, test suite reduction, and test case prioritization–to reduce size of test suite and, as an additional benefit, execution time [23, 19]. The goal of our approach is not to optimise *size* of test suite but rather test suite permutation to reduce execution time. We can apply our approach to an already minimised test suite to reduce time consumed further. If testers believe that test minimisation or reduction techniques will result in reduced fault finding capability [9], our approach can be applied directly to the the full test suites. The following section describes our approach.

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

3. OUR APPROACH

To maximise temporal re-use of instructions across several executions of the program (or test case runs), we need to determine an order of test case executions such that distance between consecutive test case executions in the order is minimized while also minimizing the total distance of the order. Note that it is important to minimise the total distance additionally, since that helps pick the best among all orders that have minimal distance between consecutive test case executions, each of which is produced by a different starting vertex or test case execution in our case. This is similar to the problem of least cost Hamiltonian Path which is known to be NP-hard. In our approach, we use the nearest neighbour algorithm as an approximate solution since it effectively solves the sub-problem of minimising distance between consecutive test case executions that is important for leveraging immediate temporal locality. Distance between two test cases, T_i and T_j, is defined as

$$D(T_i, T_j) = \#instructions\ different\ between\ T_i\ and\ T_j \quad (1)$$

The rationale for this definition of distance is that instruction locality between test runs is greater when there are more common instructions between them (or fewer different instructions). For instance, let's say for test runs $T1$ and $T2$, 30% of their instructions were the same, and for $T1$ and $T3$, 65% of the instructions were the same. Then, we improve the chances of re-visiting the same instructions between two test runs if we place $T1$ next to $T3$, rather than $T2$, in the order of test execution.

To enable scalability, we use basic blocks instead of instructions to compute distance in Equation 1. $D(T_i, T_j)$ is the symmetric difference between the set of basic blocks visited by T_i and T_j. Note that, $D(T_i, T_j) = D(T_j, T_i)$ in our definition. In our implementation we express the distance as a fraction of the total number of basic blocks visited by all test cases[3], i.e.

$$D(T_i, T_j) = \frac{\#basic\text{-}blocks\ different\ between\ T_i\ and\ T_j}{Total\ \#basic\text{-}blocks\ visited\ by\ all\ tests} \quad (2)$$

As stated earlier, our approach to solve the distance minimisation problem is based on the nearest neighbour algorithm. For a sequence with N test case runs and T_p being a test case run at position p, our approach re-orders (or permutes) the sequence such that,

$$D(T_i, T_{i+1}) <= D(T_i, T_j), \quad \text{where } j > i+1 \text{ and } i \in \{1, ..., (N-2)\} \quad (3)$$

The condition in Equation 3 states that for a test case run at position i, T_i, the next test case run in the permuted sequence, T_{i+1}, should be the one that has the least distance to T_i among the test case runs that have not yet been visited (or permuted). Our algorithm is illustrated in Table 1. We provide as inputs N test cases in some given sequence with T_i being test case at position i. We also provide an input thresold distance, Thr, so that test case runs who are within Thr distance of each other will be considered *neighbours* and used in the nearest neighbour computation. Test cases whose distance exceeds Thr are not considered neighbours and will be examined for ordering only after all the neighbours are visited. Thr is a function of cache size and program size and is used as an indicator of the distance limit beyond which immediate temporal locality between test cases cannot be improved by ordering[4]. This in turn helps save computation effort and time by not having to consider test cases that exceed Thr in the nearest neighbour computation.

Our algorithm shown in Table 1 computes the distances between all test cases and stores them in a distance matrix. The heuristic we use to pick the starting test case run in our execution order is the one with most unvisited neighbours. We set this to *current* test case and mark it with a visited flag. We then check if the *current* test case has unvisited neighbours and pick the one that is closest. This becomes the new visited *current* and the process is repeated with neighbours. If there are no unvisited neighbours, and we still have test cases that are not visited, we pick a new *current* test case in the same way as we picked the starting test case in the beginning and repeat the process with neighbours. This naïve greedy algorithm for ordering test cases using nearest neighbour has a complexity $O(N^2)$, where N is the number of test cases. In our future work, we plan to reduce complexity in creating the ordering using approximation algorithms.

3.1 Implementation

For achieving step 1 of the algorithm in Table 1 we use Intel's Pin [14], a dynamic binary instrumentation framework which allows the development of analysis tools (commonly refered to as Pintools). We implemented our Pintool to record the basic blocks visited for every test case execution. The remaining steps 2 to 10 of the algorithm to generate the permuted sequences of test cases are implemented in C++11 and LLVM passes [13]. Given a C/C++ program and test cases, our implementation will execute each test case independently and dynamically analyse it with the Pintool. Each test case is mapped to a set of visited basic blocks, which is then used to compute the distance between test cases and to build the distance matrix. Finally, we use the distance matrix to generate the optimised permutation sequence. In the next section, we present the questions evaluated in our experiment along with a description of tools and subject programs used for our measurements.

4. EXPERIMENT

Table 2: Subject programs used in our experiment

Subject	Size (Avg. Exec. Instrucs.)	#Test Cases
`replace`	1.28e+04	5542
`sed`	5.36e+06	358
`tcas`	2.23e+02	1608
`totinfo`	1.89e+04	1052

In this paper, we only present a preliminary evaluation with a small number of programs. We plan to undertake an extensive evaluation of our approach with a large set of programs and test suites in our future work. The questions we seek to answer in our preliminary evaluation are,

Q 1. *Is time taken for test suite execution affected by the order in which test cases are executed?*

To answer this question, for each subject program and associated set of test cases, we first randomly permute the sequence of test cases. We perform **2000 such random permutations**[6] and measure time taken for execution. The distribution of time measurements gives an estimate of the effect of test case ordering. The effect of permuting test cases is hard to predict and can vary widely among programs (and their test cases) based on their characteristics (such as

[3]This is done so that distances can be compared against a threshold defined subsequently.

[4] Thr = 1 - (Average #instructions across test runs / Cache size in instructions) if (program size < cache size). Else, Thr is median of minimum and maximum distance observed in the distance matrix.

[6]Time needed in running the experiments was a severe limiting factor for significantly larger number of permutations.

Table 1: Algorithm for permuting sequence of test cases for improved temporal locality

Algorithm *Permute Sequence of Test Case Runs*
Input: N test cases, P program,
Thr defining cutoff distance between test cases to be neighbours.
Output: List R with the permuted sequence of N test cases.
begin
1: For $1 \leq i \leq N$, run each test case, T_i, on P and record the set of basic blocks visited, $\{BT_i\}$.
2: Mark all test case runs as not visited (or unvisited). Initialise R to be an empty list
3: $\forall i, j \in \{1, N\}$, build a distance matrix of T_i to T_j , such that $D(T_i, T_j) = \{BT_i\} \triangle \{BT_j\}$.
4: From the distance matrix, select a starting T as the one that is not visited and has most[5] unvisited neighbours (i.e. $distance \leq Thr$).
5: Set this to $current\_T$, mark as visited, and insert it into the end of list R.
6: If $current\_T$ has no unvisited neighbours, go to Step 9.
7: Pick the neighbour that is not visited and has least distance from $current\_T$.
8: Go to Step 5.
9: If there are unvisited test case runs in matrix D, go to Step 4.
10: Output R as the permuted sequence of test cases.
end

control flow, memory references, computations, number of instructions).

Q 2. *How does time consumed by our optimised permutation compare to random test case permutations?*

We measure the time consumed by executing test cases in the permuted sequence produced by our algorithm (referred to as *optimised permutation*). We compare it to the mean time consumed and the distribution of the random permutations measured in **Q1**. We also use the best and worst case times of the random permutations in our comparison.

4.1 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. We ensure no additional services are running on the server edition of Ubuntu when we perform measurements. We measure execution time of our algorithm and program test case runs using a *system_clock* function included in the *std* :: *chrono* library in C++11.

4.2 Subject Programs

We use 4 programs from the SIR repository [8] for our experiment. Programs include pattern matching and substitution (`replace`), stream text editor (`sed`), a statistics program (`totinfo`), and an aircraft collision avoidance system (`tcas`). The subject programs have between 358 and 5542 test cases. Program execution size (reported as average number of executed instructions across all test cases) and number of test cases for the programs used in our experiment is shown in Table 2.

For the subject programs, each test permutation is executed multiple times in our measurement.[7] This is done so that we report the execution time of all programs on a comparable scale in seconds. Executing the test permutation multiple times simply scales the execution time and has no effect on the interpretation of the results and the impact of permutations reported. The results from our experiments and their analyses is presented in the next Section.

[7]For `tcas`, each test permutation is run 1000 times. 100 times for `replace, sed, totinfo`.

5. RESULTS AND ANALYSIS

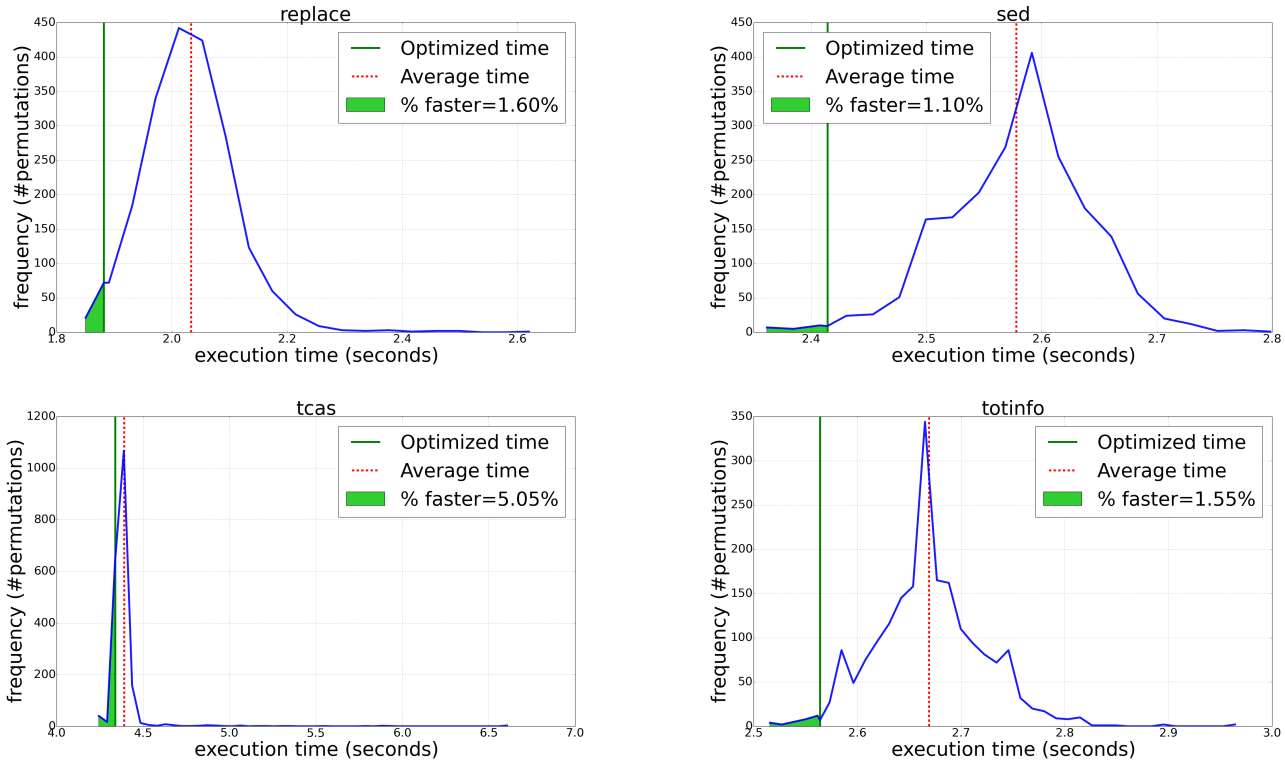
5.1 Q1 Analysis

We ran 2000 random permutations of test cases, measuring execution time of each permutation, for each of the four SIR programs. Table 3 shows, for each subject program, the histogram frequencies of execution times for the 2000 random permutations using a frequency polygon. The mean execution time across permutations is shown as a dashed red vertical line. The execution time of the optimised permutation generated by our algorithm is shown as a green line. The shaded area under the curve to the left of the green line represents the percentage of random permutations that execute faster than our optimised version. Table 4 shows the median[8], best and worst execution times in the distribution over random permutations. We also show our optimized permutation time for easy comparison. We do **not** show standard deviation, since we found that the execution times for all subject programs over the random test permutations were *not normally distributed*. We confirmed this by running a chi-squared goodness of fit test, and the p-values for all programs was 0 (rejecting the null hypothesis that they are normally distributed at 5% significance level).

It can be seen from the plots in Table 3 that execution times clearly vary across test permutations. The extent to which execution times vary is different for each program and associated tests. For instance, in program `sed`, execution times are symmetrically distributed. The execution times for the random permutations lie close to the mean for `sed`. We believe this is because the size of a single test execution over `sed` exceeds the L1 instruction cache size. Thus permutations will have limited effect on instruction locality across test executions, or execution time as a result of that, at the L1 level. They may, however, still have an effect on the L2 level cache. Distribution of execution times in the other subject programs are positively skewed with long tail ends on the right. This is especially prominent for the programs - `tcas, replace`. The differences between the best and worst permutation execution times were 57.1% and 44.3%, respectively. We believe the large difference is because test case distances were distributed over a wide range for these two programs. Test suites for these programs were such that there were clusters of test cases with low distances within the cluster, i.e. they execute similar control flow paths. Distances between test cases across clusters were high. As a result, random permutations that change the ordering of test cases within a cluster will have little effect on the instruction

[8]The median and mean for all subject programs were the same numerically up to the second decimal place. As a result, we only use one of them, mean, in our comparisons.

Table 3: Histogram frequencies of execution time for random permutations for 4 SIR programs (optimised permutation is also shown)



locality. On the other hand, permutations that changed the order across clusters will have a negative effect on instruction locality. The size and number of clusters will determine the size of effect.

Q1 Answer.
It is clear from the plots in Tables 3 that *the order in which test cases are executed* ***affects*** *execution time.* The nature of the program and test cases, in terms of size of a single test execution and the range of distances between test cases, determine the magnitude of the effect. The differences between worst and best permutation execution times ranged from 18.3% to 57.1% across our subject programs.

Table 4: Statistics on execution time (in secs) distribution compared with optimised permutation (Opt.)

Subject	Median	Worst	Best	Opt.
`replace`	2.03	2.64	1.83	1.88
`sed`	2.58	2.81	2.35	2.41
`tcas`	4.39	6.63	4.22	4.33
`totinfo`	2.67	2.97	2.51	2.56

5.2 Q2 Analysis

The green vertical line in the plots in Tables 3 shows the execution time of our optimised permutation. It is worth noting that we took 100 measurements of the execution time of our optimised permutation for each program. The measurements were only marginally different (same numerically up to the third decimal place) and as a result, we only show a vertical line in the plots. The 'Opt.' column in Table 4 shows the average over these 100 time measurements of the optimised permutation. As stated earlier, the shaded area in the plots represents the percentage of random permutations that execute faster than our optimised version. The optimised permutation does better than 97% of the random permutations on 3 of the 4 subject programs, `replace, sed, totinfo`. For `tcas`, we outperformed 93% of the random permutations. As observed earlier, test suites for the subject programs have clusters of test cases with low distances within, and high distances across clusters. Our approach for permutation ensures that test cases in clusters are executed together, effectively leveraging the instruction locality between them. We believe this is the primary reason for outperforming such a large majority of the random permutations. The sizes of these programs and numbers of test cases vary widely, as seen from Table 2. The improvement over average random permutation execution time was in the range of 1.4% to 7.4% for these programs. The improvement over the worst permutation was significant, ranging from 13.8% to 34.7%. The best permutation was comparable to our optimised permutation and was faster by a small margin, 1.9% to 2.6%.

Q2 Answer.
We found that our optimised permutation outperformed a significant fraction (more than 93%) of random permutations over all subject programs. The gain in execution time varied across programs. It was in the range of 1.4% to 7.4% over average, and 13.8% to 34.7% over worst permutation.

6. CONCLUSION AND FUTURE WORK

We proposed an approach for reducing time consumed during the execution of a test suite by permuting test cases for improved instruction locality. The idea in this paper of optimising locality of references *across* program runs, rather than just a single run is entirely novel. The approach we use to improve instruction locality, aims to minimise the distance between neighbouring test cases in the execution order. Distance is measured as the number of different instructions executed between two test cases. We use a greedy approximation algorithm for permuting the test cases based on their distance.

We conducted a preliminary evaluation using four subject programs from the SIR repository. We used 2000 random permutations of the test suite in each of four subject programs. The results in our experiment showed that the order of test case execution **matters** for execution time. The differences between worst and best permutation running time ranged from 18.3% to 57.1%, across our subject programs. Our optimised permutation outperformed a significant fraction of random permutations over all subject programs. The gain in execution time varied across programs. It was in the range of 1.4% to 7.4% over average, and 13.8% to 34.7% over worst permutation. As with compiler optimisation techniques, the effectiveness of our approach depends on the characteristics of the program and test cases. Size of the program, distances between test case runs, number of test cases, cache size, will all have a significant effect on the time savings from our approach.

The initial experiment in this paper provides evidence that, (1)Test case execution order is important to consider for execution time, and (2)Improving instruction locality across test case executions helps improve execution time. These two observations are the main contributions in this paper. We are aware that there may exist better algorithms to generate an optimised permutation than the one used in this paper. We will explore these along with scalability challenges, with respect to program size (that exceeds cache size) and number of test cases, in our future work. Once we address the scaling challenges with respect to program size and number of tests, we will explore large applications from different domains with nightly (or some frequent periodic) test suite runs.

7. REFERENCES

[1] Polly LLVM library. http://polly.llvm.org/index.html.

[2] Kristof Beyls and Erik D'Hollander. Reuse distance as a metric for cache behavior. In *Proc. of the IASTED Conf. on Parallel and Distributed Computing and Systems*, volume 14, pages 350–360, 2001.

[3] Kristof Beyls and Erik D'Hollander. Refactoring for data locality. *Computer*, 42(2):62–71, 2009.

[4] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. *Compiler optimizations for improving data locality*, volume 28. ACM, 1994.

[5] J Bradley Chen and Bradley DD Leupen. Improving instruction locality with just-in-time code layout. In *Proceedings of the USENIX Windows NT Workshop*, pages 25–32, 1997.

[6] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44(5):609–623, 1995.

[7] Peter J Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

[8] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[9] Mats PE Heimdahl and Devaraj George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 176–185, 2004.

[10] W-m W Hwu and Pohua P Chang. Achieving high instruction cache performance with an optimizing compiler. In *ACM SIGARCH Computer Architecture News*, volume 17, pages 242–251. ACM, 1989.

[11] Teresa L Johnson, Matthew C Merten, and Wen-Mei W Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 57–64, 1997.

[12] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 357–368. IEEE Computer Society, 1998.

[13] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[14] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[15] C. Pyo, K. Lee, H. Han, and G. Lee. Reference distance as a metric for data locality. In *High Performance Computing on the Information Superhighway, 1997. HPC Asia'97*, pages 151–156. IEEE, 1997.

[16] Ajitha Rajan. Coverage metrics to measure adequacy of black-box test suites. In *21st ASE*, pages 335–338. IEEE, 2006.

[17] A. Ramirez, L. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, G. Lowney, and M. Valero. Code layout optimizations for transaction processing workloads. In *ACM SIGARCH Computer Architecture News*, volume 29, pages 155–164. ACM, 2001.

[18] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN Notices*, volume 33, pages 38–49. ACM, 1998.

[19] Patrick J. Schroeder and Bogdan Korel. Black-box test reduction using input-output analysis. In *ISSTA*, pages 173–177. ACM, 2000.

[20] V. Tiwari, S. Malik, A. Wolfe, and M.T.-C. Lee. Instruction level power analysis and optimization of software. In *VLSI Design, 1996. Proceedings., Ninth International Conference on*, pages 326–328, Jan 1996.

[21] N. Vijaykrishnan, M. Kandemir, M.J. Irwin, H.S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 95–106, June 2000.

[22] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *ACM Sigplan Notices*, volume 26, pages 30–44. ACM, 1991.

[23] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. 22(2):67–120, 2012.