Computing and Informatics, Vol. 40, 2021, 543–574, doi: 10.31577/cai_2021_3_543

REDUCER: ELIMINATION OF REPETITIVE CODES FOR ACCELERATED ITERATIVE COMPILATION

Hameeza Ahmed, Muhammad Ali Ismail

Department of Computer and Information Systems Engineering
NED University of Engineering and Technology
Karachi, Pakistan
e-mail: {hameeza, maismail}@neduet.edu.pk

Abstract. Low Level Virtual Machine (LLVM) is a widely adopted open source compiler providing numerous optimization opportunities. The discovery of the best optimization sequence in this large space is done via iterative compilation, which incurs substantial overheads, especially for big data applications operating on high volume and variety datasets. The large search space is mostly comprised of identical codes generated via different optimizations. However, no mechanism is implemented inside the LLVM compiler to suppress the redundant testings. In this regard, this paper proposes REDUCER for eliminating the identical code executions by performing Intermediate Representation (IR) level comparisons. REDUCER has been tested using the well-accepted MiCOMP technique in LLVM 3.8 and 9.0 compiler, with embedded (cBench) and big data workloads. In comparison to MiCOMP 19.5 k experiments, REDUCER lowers the experiment count up to 327, i.e. 98 %, and on average to 4 375, i.e. 77 %, for cBench (LLVM-3.8). Similarly, for LLVM-9.0 the reductions are up to 1 931, i.e. 90 %, and on average 5 863, i.e. 69.9 %. Due to the significant experiment reduction, for embedded workloads, the iterative compilation is up to 58.6× and on average 4.1× faster with REDUCER (LLVM-3.8) than MiCOMP, whereas, with REDUCER (LLVM-9.0) the compilation is up to 8.5× and on average 2.9× faster. Moreover, REDUCER is found to be scalable and efficient for big data workloads where the iterative compilation is reduced to few days, as code is compared one time only for a single application tested on multiple datasets.

Keywords: Iterative compilation, code redundancy, LLVM, IR, big data

1 INTRODUCTION

Low Level Virtual Machine (LLVM) [1] is an open source compiler infrastructure, which is widely adopted due to high *ease of use*, *flexibility*, *portability*, and *modularity* [1, 24, 23]. It features *source* and *target* independent Intermediate Representation (IR) code, which allows numerous optimizations to be easily applied. Among the millions of available optimizations inside LLVM, the suitable optimizations for the given *application*, *environment* (architecture, OS, compiler), and *dataset* combination can be found by repeated execution of the program with each optimization, commonly known as *iterative compilation*. The testing of this huge search space for varying *application*, *dataset*, and *environment* incurs significant time and resource overheads. These costs are especially exaggerated if the optimizations are tested for *big data* applications [7], involving high *volume* and *variety* datasets [12]. In the case of *volume*, each optimization sequence is executed with a large data size which is more time-consuming than routine sizes. Similarly, in the case of *variety*, each optimization is iteratively executed with multiple datasets, increasing the number of runs.

Several techniques [14, 15, 3, 10, 11] have been proposed for reducing the search space in LLVM. However, these techniques consider the identical codes as separate optimization, hence increasing the overall search space due to redundant testings. Although [20, 22, 19], emphasizes on reducing the search overhead by detecting identical codes. But, no such method has been practically adopted by LLVM compilation techniques, for detecting identical optimization sequences. Instead, iterative compilation is treated as a black box, and complex techniques are proposed for reducing the search space.

The repeated code execution makes the process of iterative compilation infeasible by unnecessarily increasing the number of experiments which ultimately leads to heavy resource and time wastage. The overheads are especially inflated for big data applications processing high volume and variety datasets [7, 12]. For instance, an application has total 19.5 k optimized codes, with only 500 unique codes. If the application is tested with large size data taking approx. 3 h, then $19.5\,\text{k} * 3 = 58\,500$ h approx. is required to find the best optimization sequence, having 19 k redundant experiments. Whereas, if the unique 500 codes are detected at the start, then only $500 * 3 = 1\,500$ h + detection time is needed to search the best optimization. Similarly, for testing the considered application with 5 varying datasets each taking 2 h, approx. $19.5\,\text{k} * 2 * 5 = 195\,000$ h is required with 19 k redundant tests. However, if the redundant tests are suppressed, then only $500 * 2 * 5 = 5\,000$ h + detection time is enough.

For reducing the LLVM optimization space, this paper proposes **REDUCER**, which lowers the search space by detecting *identical* codes. It has been tested using well-accepted Mitigates the Compiler Phase-ordering (MiCOMP) [3] technique in Low Level Virtual Machine (LLVM) compiler [1]. LLVM makes the REDUCER portable enough to work on any host platform as the reductions are made on the basis of machine-independent code. REDUCER selects the executable candidates

after comparing the generated IR with the already existing IR codes. In case, only if the IR does not match with existing IRs, it is selected as an executable candidate. REDUCER testing has been performed using embedded, i.e. cBench benchmark suite (LLVM-3.8 & 9.0), and big data workloads (LLVM-9.0). MiCOMP [3] was tested in 2017 on LLVM-3.8[1], using *agglomerative* clustering, thus for LLVM-9.0 we have extended MiCOMP approach by finding the optimization clusters, but using k-means algorithm. It has been observed that our derived k-means based sub-sequences for LLVM-9.0 exploit greater speedup than MiCOMP's *agglomerative* based [3] sub-sequences for LLVM-3.8.

For both LLVM 3.8 and 9.0, REDUCER shows substantial reduction in experiment count of embedded workloads. Also, it is discovered that the increase in optimization sequence length, increases the redundancy fraction, which encourages the testing of longer sequence lengths. In this manner, the larger optimization space can be exploited, which has been uncovered till now. Moreover, Dynamic Programming (DP) has been applied to estimate the unique code sequences. DP is found to be less accurate than REDUCER, with increased experiment count. Despite a significant number of code comparisons, REDUCER is observed to be faster than MiCOMP, and this speed is expected to significantly increase for longer sequences and big data applications. As evident via experiments, REDUCER cuts down the iterative compilation of big data benchmarks to a few days in comparison to months and years taken by MiCOMP. This way, the possibilities of finding the best optimization sequence for *big data* applications are enlarged due to REDUCER. Hence, REDUCER is a simple yet effective solution to be adopted by any iterative compilation technique. Following are the main contributions of this paper:

1. To the best of our knowledge, the first work which practically accelerates the iterative compilation process of LLVM by reducing the search space via simple IR code comparisons.
2. Higher portability, i.e., REDUCER can work on any host platform due to machine-independent LLVM IR code.
3. Exploitation of greater speedup by extending MiCOMP for LLVM-9.0 using k-means clustering.
4. Facilitating the iterative compilation process for *big data* applications, i.e., repeated tests are suppressed by comparing code one time for a single application tested on multiple datasets.
5. Facilitating the exploitation of large search space via longer sequence length, because the analysis shows redundancy fraction is increased with sequence length.

Rest of the paper is organized as follows: Section 2 discusses background and motivation, REDUCER is presented in Section 3, the experimental setup is discussed

[1] LLVM-3.8 was the newer version in 2017, but at the time of experimentation of this paper, i.e. 2020, LLVM-9.0 is the newer version.

in Section 4, Section 5 analyzes the results. Section 6 discusses the related work, followed by a conclusion in Section 7.

2 BACKGROUND AND MOTIVATION

This section discusses various terminologies and concepts. Also, the motivation behind this work is presented.

2.1 Compiler

Compiler is a program that translates the high-level language code into architecture specific assembly and enables the optimizations for exploiting the hardware resources. This implies that despite the presence of powerful hardware design, the performance goals are not met due to a lack of competent software solutions. In the present era, the hardware resources (processors, caches, DRAMs, and hard disks) show reliance on the compiler for extracting the higher performance, energy efficiency, and reduced development time. The compilation life cycle proceeds by passing the source code through *front end*, *middle end* (optimizer), and *back end*. The front end emits Intermediate Representation (IR) code, which is passed through middle end to perform specific optimizations like *inlining*, *unrolling*, etc. In the end, the back end generates the machine code [25, 18, 8, 9, 2, 17].

2.2 LLVM

Low Level Virtual Machine (LLVM) is an open source compiler infrastructure, containing reusable and modular compiler technologies. It provides wide optimization opportunities due to library based optimizer's design. Besides, it allows flexibility as the optimizations passes can be ordered to be executed in a specific order. This way, the design enables the selection of individual optimization passes to execute. LLVM allows working with anyone optimizer separately, without considering other modules attached to it. Whereas, the traditional compilers are designed as tightly interconnected code, which is tougher to break into small parts for better understanding and use. The LLVM code is represented by Static Single Assignment (SSA)-based Intermediate Representation (IR), which provides *low level operations*, *type safety*, *portability*, *flexibility*, etc. The LLVM IR appears to be a universal IR, as all the phases of LLVM compilation use this IR [1, 24, 23].

2.3 Iterative Compilation

For an *application*, *dataset*, and *architecture*, the optimal set of optimizations is found via *iterative compilation*, where the best optimization combination is detected by running a program multiple times, each time with different optimizations combinations. This iterative testing involves billions of different optimizations. To avoid

this huge space exploration, standard optimizations, i.e., -O1, -O2, -O3, -Os, have been provided in commercial compilers, which on average bring good performance on a set of applications. However, there exist optimizations combinations that outperform the standard optimization levels for many programs by a considerable margin. Finding the best optimizations ordering can significantly improve the performance metrics like *execution time*, *energy*, *power consumption*, and *code size*.

Despite the great potential offered by iterative optimization, it is not widely used in compilers, because it requires numerous recompilations and training runs to detect the best optimization combination for a given program. This way, the costly overhead of recompilation and training runs can eradicate the benefits of iterative optimization, hence it is not a feasible option due to excessive compilation time [3, 4, 12, 6].

2.4 Phase Ordering Problem

In multi-phase optimizing compilers, there exist no ideal ordering of phases which results into *phase ordering* problem. For instance, a transformation pass X, optimizes the code such that the effect of some optimizations to be performed by the following pass Y is hindered. Similarly, by switching phases order, pass Y can deprive pass X optimizations. On the contrary, a phase can bring new optimization opportunities for the other. In this situation, it is the responsibility of compiler writers to carefully consider the order in which each optimization phase is performed [4, 3].

Consider an application that is passed through the front end by disabling all optimizations to emit an Intermediate Representation (IR) a. Consider a set of optimizations $o_1, o_2, \ldots, o_n$. For finding the suitable optimization for application, a is required to be passed through the optimization set. The optimizations space due to the phase-ordering problem is in the factorial as permutations are involved, which is represented by Equation (1). Where n is the number of optimizations under study [4, 3].

$$|\Omega_{Phases}| = n!. \tag{1}$$

Considering the optimizations to be applied repeatedly with a variable-length sequence of optimizations. The problem space will be expanded as per Equation (2). Where, m is the maximum desired length for the optimization sequence [4, 3].

$$|\Omega_{Phases\_Repetition\_variableLength}| = \sum_{i=0}^{m} n^i. \tag{2}$$

Even with reasonable n and m, the optimization search space is huge. For instance, with n and m 10, an optimization search space consisting of more than 11 billion different optimization sequences is formed [4, 3].

The phase-order search problem finds an optimal optimization sequence for a program from an infinitely huge space of optimization sequence. The problem is combinatorial in nature having no convexity or linearity properties, thus a given

sequence cannot be called optimal. Only the performance of a sequence can be compared relative to a default compiler optimization sequence (like -O3). The sequence is said to be good for a program, if it is showing a noticeable improvement in performance over default optimization sequences. Therefore, it is not necessary that a good sequence is also an optimal one [33]. Similar to previous works [3, 15, 11], this paper reports the performance speedup relative to LLVM's highest optimization level of -O3.

2.5 MiCOMP

Several techniques [14, 15, 11, 33] have been proposed for search space reduction in the given scenario, but the Mitigates the Compiler Phase-ordering (MiCOMP) has been selected in this paper, due to its *systematic* and *reproducible* approach for reducing the optimization space of those compilers, which exhibit the phase ordering problem. It works by clustering the LLVM's -O3 optimization passes into different groups (sub-sequences). The optimizations order within a group is internally fixed, but the group ordering can be altered.

The phase-ordering is exploited by using the sub-sequences instead of individual optimizations, which reduces the search space significantly. These sub-sequences can be found using any automated clustering technique. MiCOMP is effective as it exploits greater speedup by testing smaller search space as compared to other techniques. MiCOMP reduces the search space (Equation (2)) by fixing n to be 5 and m ranges from 3 to 7 [3]. For the experimentation of MiCOMP, n and m are fixed to 5 and 6, respectively, in [3], as shown by Equation (3):

$$\Omega = \sum_{i=0}^{6} 5^i = 19.5\,\text{k}. \tag{3}$$

Assuming $m = 6$, $n = 63$ (LLVM-3.8), Equation (2) becomes

$$\Omega = \sum_{i=0}^{6} 63^i = 62.5\,\text{b}. \tag{4}$$

Hence, MiCOMP achieves a speedup of $62.5\,\text{b}/19.5\,\text{k} = 3\,201.2\,\text{k}$ approx. over LLVM-3.8[2]. Amongst the given 19.5 k tests, there exists a strong likelihood of *identical* codes, hence such codes are not required to be executed repeatedly. However, identical code testing has not been performed by MiCOMP. Assuming α be the fraction of identical codes out of 1, for the given optimization space, Equation (2) becomes

$$\Omega = \sum_{i=0}^{m} n^i * (1 - \alpha). \tag{5}$$

[2] LLVM-3.8 -O3 has 63 internal passes.

Assuming $\alpha = 20\,\% = 0.2$, MiCOMP Equation (3) becomes

$$\Omega = \sum_{i=0}^{6} 5^i = 19.5\,\mathrm{k} * (1 - 0.2) = 15.6\,\mathrm{k}. \tag{6}$$

In this manner, for $\alpha = 0.2$, MiCOMP search space is reduced by 20 % by ignoring redundant optimizations. Let $P(\alpha)$ be the probability of finding α percent repeated codes, where $0 \leq P(\alpha) \leq 1$. $P(\alpha)$ depends on input application characteristics and the effect of optimizations on that. It is independent of platform features for generic codes.

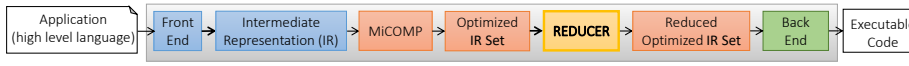


Figure 1. Compilation flow

3 REDUCER

For suppressing the identical codes in LLVM, this paper proposes REDUCER as shown in Figure 1. As it can be observed, an application is compiled by front end, which emits Intermediate Representation (IR), then the IR is passed through MiCOMP's given optimization sub-sequences and REDUCER, which emits unique optimized IR by comparing each new IR with existing old IR. As represented via Equation (7), the un-optimized IR (x_1) is passed through optimization set (opt_n) to get the new IR (y_n). The new IR (y_n) is only retained if it is not identical to old IRs (y_o), otherwise, the IR is discarded. The execution of REDUCER is continued until redundancy checking is not done for all the optimized IRs. Once the REDUCER is stopped, the found unique code IRs are added to the reduced optimized IR set. In the end, the reduced set is passed through the backend for converting the IR codes into target-specific executables.

$$y_n = opt_n(x_1), \text{ if } y_n \neq y_o. \tag{7}$$

The overall compiler optimization space reduction is shown in Algorithm 1. The algorithm receives unoptimized IR, -O3[3] optimized IR, -O3's optimization sequence, the desired number of clusters, and maximum sequence length as input. Firstly, MiCOMP procedure is invoked for constructing the required clusters using a clustering algorithm. Then, a *sequence set* is constructed by inserting the appropriate optimization permutations of length 1 to maximum sequence, which are generated from the set of derived optimization clusters.

After this REDUCER is invoked for generating the optimized IR codes by suppressing redundancies. Firstly, an -O3 optimized IR is added to the optimized set,

[3] -O3 is a baseline to compare optimization sub-sequences performance [3].

Algorithm 1: Compiler optimization space reduction

Input: Un-optimized IR (x_1), O3 optimized IR (x_2), Set of LLVM -O3 optimization sequence $o = \{o_1, \ldots, o_N\}$, Desired number of clusters (*NumClust*), Maximum sequence length (*MaxSeqLen*)

Output: Reduced Executables Set (y)

```
   /* Finding Optimal -O3 Sub-Sequences using MiCOMP                    */
 1 Construct an optimization dependency graph G = (V, E) using o;
 2 Construct a weighted adjacency matrix M from G;
 3 clusters ← M. ApplyClusteringTechnique (NumClust);
 4 for SeqLen in 1 to MaxSeqLen do
 5 |   SeqSet += GeneratePermutations(clusters, SeqLen);
 6 end
   /* Reducing search space by evicting repeated codes using proposed
      REDUCER                                                          */
 7 IRSet.Add(x2);
 8 for flag in SeqSet do
 9 |   temp ← x1.ApplyOptimization(flag);
10 |   if !temp.IsEquivalent(IRSet) then
11 |   |   IRSet.Add(temp);
12 |   end
13 end
   /* Compile IR codes to generate executables                         */
14 for k in IRSet do
15 |   y ← CompileIRtoExecutable(k);
16 end
```

and then a new optimized IR is generated by applying the individual optimization subsequence obtained from the *sequence set*. It is followed by comparing the generated subsequence IR with existing -O3 optimized IR. In the case of different codes, the generated IR is added to the optimized set, otherwise, it is discarded. This process is repeated for all optimization sub-sequences. Each new IR is compared with the ones generated in previous iterations. Finally, the reduced set of IR codes is compiled to generate executables.

The proposed REDUCER algorithm is based on the sequential comparison, which is time-consuming process. However, this timing overhead is justified because it eliminates redundant testing, which is highly beneficial for *big data* applications processing *volume* and *variety* datasets. For a single application, the IR comparison is done one time only, irrespective of the size and format of datasets. Consider an application, operating on 5 datasets with average execution time as d_1 (5 min), d_2 (10 min), d_3 (25 min), d_4 (30 min), d_5 (65 min). With MiCOMP overall time is roughly $(5 + 10 + 25 + 30 + 65\,\text{min} = 135\,\text{min} * 19\,531 = 2\,636\,685\,\text{min})$. However, with the inclusion of REDUCER having $\alpha = 0.8$, and comparison time = 2 880 min, the overall time is roughly $2\,880 + (135 * 3\,907)\,\text{min} = 530\,325\,\text{min}$, which is around 4.9× faster than MiCOMP.

Parameters	**Embedded Workloads**	**Big Data Workloads**
Total RAM	8 GB	64 GB
Total Swap	2 GB	2 GB
Disk Cache	1 GB	1 GB
Model Name	Intel(R) Core(TM) i7-8550U CPU @ 1.80 GHz	Intel(R) Xeon(R) Silver 4216 CPU @ 2.10 GHz
Page Size	4 kB	4 kB
Hard Disk	1 TB SATA Harddisk	1 TB SATA Harddisk
Operating System	Linux Ubuntu 18.04.4 LTS	Linux Ubuntu 18.04.4 LTS
L3 Cache	8 192 KiB Associativity: 16-way Set-associative	8 192 KiB Associativity: 16-way Set-associative
L2 Cache	256 KiB Associativity: 4-way Set-associative	256 KiB Associativity: 4-way Set-associative
L1I, D cache	32 KiB Associativity: 8-way Set-associative	32 KiB Associativity: 8-way Set-associative
Compiler	LLVM-3.8, LLVM-9.0	LLVM-9.0
Benchmark	Ctuning cBench suite v1.1 [13, 5, 16] dataset one	Rodinia [28], Phoenix [31], Cortex Suite [30], Genann [32], Grep [29]
LLVM-9.0	k-means (5 clusters), Python-3.8.1 scikit-learn	Same

Table 1. Experimental setup

4 EXPERIMENTAL SETUP

This section discusses the details of the setup which has been established to test the proposed technique. Firstly, the steps behind finding the optimization clusters for LLVM-9.0 are discussed. Then, the performance benchmarks and metrics are mentioned. Finally, the implementation steps of REDUCER are discussed.

4.1 LLVM 9.0 Clusters

k-means clustering is a partitioning method that tries to discover the k number of clusters. The algorithm specifies the cluster centroid as the mean of the points. Firstly, k is selected randomly of the objects in the data set, each of which represents a cluster mean. For each of the remaining objects, an object is allocated to the cluster, on the basis of shortest Euclidean distance between the cluster mean and the object. Then, the algorithm iteratively improves the within-cluster variation. For each cluster, the new mean is computed using the objects allocated to the cluster in the previous iteration. Finally, all the objects are reassigned using the updated means as the new cluster centers. The iterations continue until the clusters built in the current turn are the same as the previous turn [36, 37].

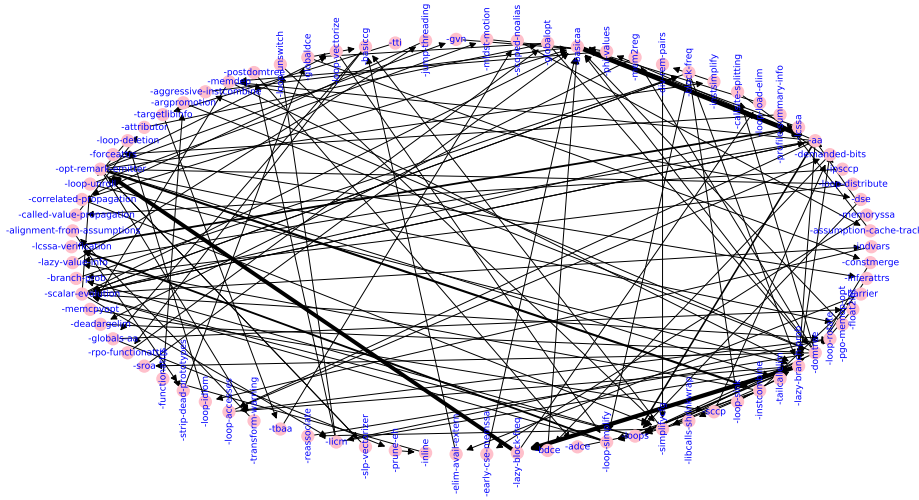


Figure 2. Directed graph for LLVM's 9.0 -O3. Each node represents an optimization pass, edge thickness depicts the strength in the connection between two nodes.

Using MiCOMP [3] approach (Algorithm 1), clusters of size 5 have been found using well-accepted *elbow* method[4] [36] for LLVM-9.0 -O3 optimization sequence via k-means technique in Python, as mentioned in Table 1. The directed graph is shown in Figure 2. The obtained clusters for LLVM-9.0 are presented in Table 2. In comparison to MiCOMP implementation in [3], we believe our implementation is easier, adaptable, and reproducible as it is done using basic k-means technique via Python based library. Whereas, in [3] MATLAB is used with a complex Graph Agglomerative Clustering (GAC) toolbox [26], which is not easily adaptable for producing the results. Our k-means based clusters exploit better performance than GAC as evident by Section 5.4.1. It is possibly because all merges are final in agglomerative clustering that is once a decision is made to combine two clusters it cannot be undone afterward, which prevents a local optimization criterion from becoming a global optimization criterion. This creates difficulty for high-dimensional, noisy, and complex graph data with multiple edges like Figure 2. This issue is tackled by partitioned based k-means clustering technique. Hence, k-means appears to be the suitable choice in a given situation [37].

4.2 Benchmark and Performance Metrics

For evaluating the proposed technique, embedded workloads belonging to automotive, security, office, and telecommunication categories from Collective Benchmark

[4] It chooses the optimal number of clusters by fitting the model for a range of a number of clusters k values [36].

Sub-seq	Compiler Passes (LLVM-3.8) [3]	Our derived Compiler Passes (LLVM-9.0)
A	-ipsccp -globalopt -deadargelim -simplifycfg -functionattrs -argpromotion -sroa -jump-threading -reassociate -indvars -mldst-motion -lcssa -rpo-functionattrs -bdce -dse -inferattrs -prune-eh -alignment-from-assumptions -barrier -block-freq -loop-unswitch -branch-prob -demanded-bits -float2int -forceattrs -loop-idiom -globals-aa -gvn -loop-accesses -loop-deletion -loop-unroll -loop-vectorize -sccp -strip-dead-prototypes -inline -globaldce -constmerge	-forceattrs -inferattrs -callsite-splitting -ipsccp -called-value-propagation -attributor -globalopt -mem2reg -deadargelim -lazy-block-freq -prune-eh -inline -functionattrs -argpromotion -memoryssa -jump-threading -libcalls-shrinkwrap -branch-prob -reassociate -loop-simplify -lcssa-verification -loop-rotate -indvars -loop-idiom -loop-deletion -mldst-motion -gvn -memcpyopt -sccp -dse -barrier -float2int -loop-distribute -loop-vectorize -slp-vectorizer -alignment-from-assumptions -strip-dead-prototypes -constmerge -instsimplify
B	-licm -mem2reg	-lazy-branch-prob -block-freq -licm -loop-unroll -demanded-bits -loop-accesses -loop-sink
C	-loop-rotate -instcombine -loop-simplify	-instcombine -simplifycfg -tailcallelim -loop-unswitch -adce -div-rem-pairs
D	-memcpyopt	-sroa -early-cse-memssa -correlated-propagation -aggressive-instcombine -pgo-memop-opt -lcssa -scalar-evolution -phi-values -bdce -loop-load-elim
E	-loop-unswitch -adce -slp-vectorizer -tailcallelim	-globals-aa -elim-avail-extern -rpo-functionattrs -globaldce

Table 2. Compiler optimizations clusters using MiCOMP for LLVM-3.8 -O3 [3] and our derived for LLVM-9.0 -O3

(cBench) programs [13, 5, 16] are used, as described in Table 3. The evaluation is done in terms of *percentage experiment reduction*, *percentage redundancy fraction*, *speedup*, and *percentage time improvement* metrics represented by Equations (8), (9), (10), and (11).

$$\text{Percentage Experiment Reduction} = \frac{\text{old count} - \text{new count}}{\text{old count}} * 100, \tag{8}$$

$$\text{Percentage Redundancy Fraction} = \frac{\text{Redundant Codes Count}}{\text{Total Codes Count}} * 100, \tag{9}$$

$$\text{Speedup} = \frac{\text{Execution Time}_{base}}{\text{Execution Time}_{new}}, \tag{10}$$

$$\text{Percentage Time Improvement} = \frac{\text{Execution Time}_{base} - \text{Execution Time}_{new}}{\text{Execution Time}_{base}} * 100. \tag{11}$$

4.3 REDUCER Implementation Details

REDUCER has been implemented using *bash script* in Linux with 5 optimization clusters and a maximum sequence length of 6. The implementation is inspired from [22] by comparing the checksum of each IR code with the ones stored in a file. In case, if checksums are not matched, the new code checksum is stored in the file, otherwise, it is discarded. The checksum has been computed using Linux *md5sum*

cBench Programs	Description
automotive_bitcount	Bit counter
automotive_qsort1	Quick sort
automotive_susan_c	Smallest Univalue Segment Assimilating Nucleus Corner
automotive_susan_e	Smallest Univalue Segment Assimilating Nucleus Edge
automotive_susan_s	Smallest Univalue Segment Assimilating Nucleus S
bzip2d	Burrows Wheeler compression algorithm
bzip2e	Burrows Wheeler compression algorithm
consumer_jpeg_c	JPEG compression kernel
consumer_jpeg_d	JPEG decompression kernel
consumer_lame	MP3 encoder
consumer_mad	MPEG audio decoder
consumer_tiff2bw	convert a color TIFF image to gray scale
consumer_tiff2rgba	Convert a TIFF image to RGBA space
consumer_tiffdither	Convert a TIFF image to dither noisespace
consumer_tiffmedian	Convert a color TIFF image to create a TIFF palette file
network_dijkstra	Dijkstra's algorithm
network_patricia	Patricia Trie data structure
office_ispell	Spelling checker
	Text to speech synthesis program
office_stringsearch1	Boyer-Moore-Horspool pattern match
security_blowfish_d	Symmetric-key block cipher Decoder
security_blowfish_e	Symmetric-key block cipher Encoder
security_pgp_d	Pretty Good Privacy decryption algorithm
security_pgp_e	Pretty Good Privacy encryption algorithm
security_rijndael_d	AES algorithm Rijndael Decoder
security_rijndael_e	AES algorithm Rijndael Encoder
security_sha	NIST Secure Hash Algorithm
telecom_adpcm_c	Intel/dvi adpcm coder/decoder Coder
telecom_adpcm_d	Intel/dvi adpcm coder/decoder Decoder
telecom_CRC32	32 BIT ANSI X3.66 crc checksum files
telecom_gsm	GSM for voice encoding/decoding

Table 3. cBench benchmark suite details [5, 16]

command[5]. REDUCER source code has been released on *Github*[6]. The embedded workloads have been run on Intel Core i7 laptop machine with 8 GB RAM, while big data workloads have been run on Intel Xeon Server machine with 64 GB RAM. Both machines have used the same Linux Ubuntu operating system. Further, experimental setup details are shown in Table 1.

[5] *md5sum* uses the MD5 algorithm for printing a 32-character checksum of the given file. A checksum is a string of letters and numbers used to uniquely identify a file.

[6] `https://github.com/hameeza/REDUCER/`

5 RESULTS ANALYSIS

This section analyzes the results in four parts. Firstly, a reduction in experiment count is reported, which is followed by studying the longer sequences exploitation and dynamic programming (DP) analysis. Finally, REDUCER performance is analyzed for embedded and big data workloads.

5.1 Experiment Count Reduction

REDUCER experiment count has been compared with MiCOMP's static 19.5 k[7] via Figures 3 and 4. For all applications and both versions of the compiler, the experiment count has been reduced by a significant amount.

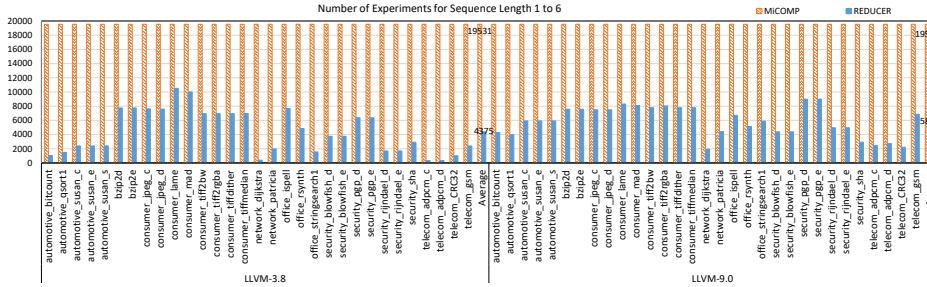


Figure 3. Number of experiments in REDUCER vs. MiCOMP for LLVM-3.8 & 9.0

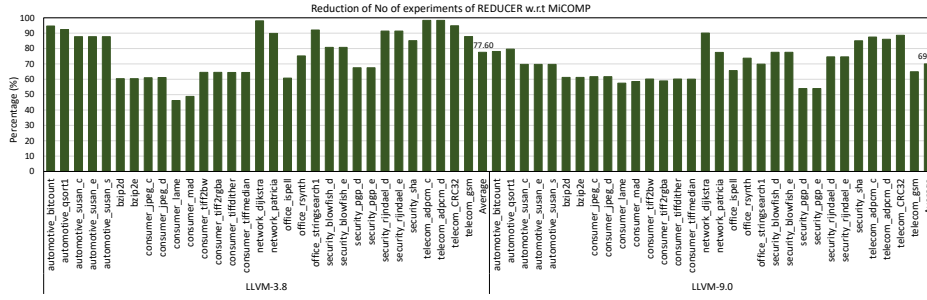


Figure 4. Percentage reduction of number of experiments in REDUCER vs MiCOMP for LLVM-3.8 & 9.0

For embedded workloads (cBench) compilation in LLVM-3.8, it can be observed that REDUCER narrows down the experiment count to 327 from MiCOMP's 19.5 k for *telecom_adpcm_c* and *telecom_adpcm_d*. This large improvement of 98 %

[7] Computed in Equation (3), keeping optimization clusters $= n = 5$ and maximum sequence length $= m = 6$.

is achieved because REDUCER detects 98 % of repeated codes produced by MiCOMP's optimization sub-sequences. Despite varying optimizations, the identical codes are generated because the majority of optimizations produce nil effect on the considered applications.

Whereas, for *consumer_lame* (LLVM-3.8), the experiment count is reduced to 10 503, bringing only 46 % improvement, which are applied on the considered application, thus generating the greater proportion of unique codes. On average, REDUCER brings a decent reduction of 77 % for LLVM-3.8. However, for the same applications and datasets, REDUCER average experiment reduction is lowered to 69.9 % for LLVM-9.0, as the newer optimization sub-sequences produce lesser identical codes. The highest reduction of 90 % is observed for *network_dijkstra* as the experiment count is lowered to 1931. This way, REDUCER discovers the emergence of highly redundant codes via the interaction of LLVM optimization passes.

The derived optimization sub-sequences for different compiler versions show varying code redundancy behavior despite keeping the uniform test environment. For *telecom_adpcm_c*, LLVM-3.8 optimization sub-sequences produce 98 % redundant codes, but LLVM-9.0 sub-sequences produce only 87 % repeated codes. Overall, LLVM-3.8 sub-sequences produce more redundant codes as compared to LLVM-9.0, which is possibly due to increased transformation opportunities in LLVM-9.0 as optimization passes in LLVM-9.0 are greater than LLVM-3.8. By enabling a greater number of transformations, the optimized codes are likely to differ from each other. With the higher experiment reduction in LLVM-9.0, it is expected that REDUCER will bring promising outcomes for future compilers as well.

Furthermore, Table 4 shows the equivalent optimal sub-sequences reported in [3] by MiCOMP for LLVM-3.8. For most applications, it has been found that MiCOMP optimal sub-sequences contain the identical code of the ones already generated in previous generations. For example, for *telecom_adpcm_c*, MiCOMP sub-sequence is ECDDCC (length 6), which is equivalent to EC (length 2), found at the second generation. This way, EC is enough and ECDDCC is not needed.

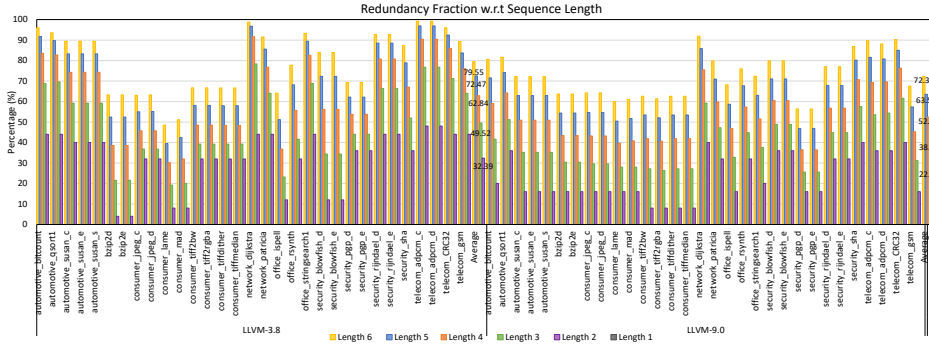


Figure 5. Application redundancy behavior w.r.t. sequence length for LLVM-3.8 & LLVM-9.0

Applications	MiCOMP Optimal Sub-Sequence	Equivalent Optimal Sub-Sequence
automotive_bitcount	BEACCA	BEACA
automotive_qsort1	CBAAAC	CBAAAC
automotive_susan_c	BDBCCB	BBCCB
automotive_susan_e	AABACA	AABACA
automotive_susan_s	ECCCDE	ECE
bzip2d	CBDACA	CBDACA
bzip2e	CBADCA	CBADCA
consumer_jpeg_c	DDC	C
consumer_jpeg_d	CCED	CED
consumer_lame	BCBACB	BCBACB
consumer_mad	DCEDCD	DCEDCD
consumer_tiff2bw	DDCAB	CAB
consumer_tiff2rgba	DDCA	CA
consumer_tiffdither	CCDCD	CDC
consumer_tiffmedian	DEDDC	EC
network_dijkstra	EECBBE	CBE
network_patricia	CECBAA	CECBAA
office_ispell	ABCBAC	ABCBAC
office_rsynth	ABCBA	ABCBA
office_stringsearch1	ABCBAC	ABCBAC
security_blowfish_d	ECEACD	CECAC
security_blowfish_e	BCCEEA	BCCEA
security_pgp_d	DCAACA	CAACA
security_pgp_e	DCA	CA
security_rijndael_d	ACCACE	ACCACE
security_rijndael_e	CAEEC	CAEC
security_sha	DACECA	ACECA
telecom_adpcm_c	ECDDCC	EC
telecom_adpcm_d	DCAACA	CAACA
telecom_CRC32	DCAACA	CAACA
telecom_gsm	DCAAC	CAAC

Table 4. MiCOMP equivalent optimal sub-sequence in LLVM-3.8

5.2 Exploitation of Longer Sequences

The effectiveness of REDUCER in facilitating the exploitation of longer sequences is shown for embedded workloads (cBench), by studying the redundancy behavior w.r.t. sequence length in Figure 5. For all applications (LLVM-3.8 & 9.0) sequence length 1, the redundancy is null. The redundancy is increased as the sequence length is increased. For *automotive_bitcount* (LLVM-3.8), redundancies are 96 %, 91 %, 83 %, 68 %, 44 %, and 0 % for sequence lengths 6, 5, 4, 3, 2, and 1, respectively. The average redundancies are 79 %, 72 %, 62 %, 49 %, 32 %, and 0 %,

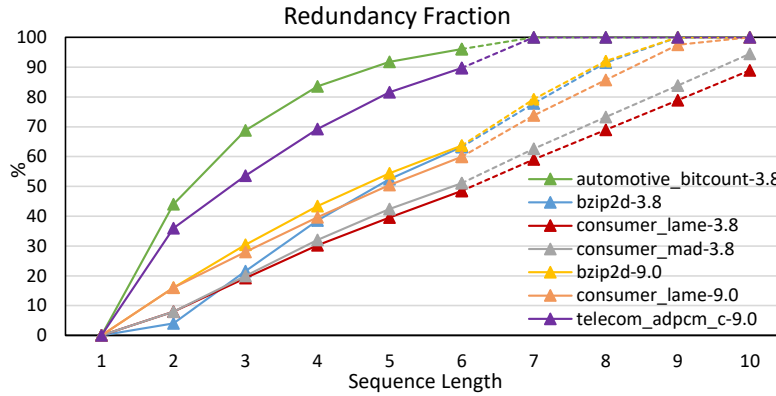


Figure 6. Redundancy fraction w.r.t. sequence length for LLVM-3.8 & LLVM-9.0, dotted lines indicate extrapolated values

respectively, for LLVM-3.8. Whereas, for LLVM-9.0, these values are 72 %, 63 %, 52 %, 38 %, 22 %, and 0 %, respectively. This study discovers how the redundant codes are increased substantially when sequence length is increased. It encourages the testing of longer sequence lengths containing optimal solutions, which are usually not exploited due to a large number of executions. With REDUCER, these longer sequences can be exploited, by the elimination of high proportion redundant codes.

The redundancy fraction has been extrapolated[8] for sequence lengths 7, 8, 9, and 10, which is shown in Figure 6 for few applications. It can be seen for *automotive_bitcount-3.8* and *telecom_adpcm_c-9.0* the expected redundancy is 100 % for length 7 and above. This way, the programmer can safely skip the longer sequences for these applications, without feeling the guilt of missing the optimal by not testing the higher space, as the fraction of unique codes is expected to be minimal in that region. On the contrary, for *consumer_lame-3.8* the predicted redundancies are 59 %, 68 %, 78 %, and 88 %, which are increased but less than 100 %. A similar trend has been observed for *consumer_mad-3.8*, *bzip2d-3.8*, and *bzip2d-9.0*. This way, longer sequence testing is needed for such applications. In this regard, REDUCER can significantly speed the testing process by suppressing the increased proportion of repeated codes in these longer sequences.

5.3 Dynamic Programming Analysis

Dynamic Programming (DP) is a recursive optimization approach that transforms a complex problem into a sequence of simpler sub-problems, and stores the solution to each sub-problem such that it is solved only once. Each time the same sub-

[8] Extrapolation is done using the Excel TREND function.

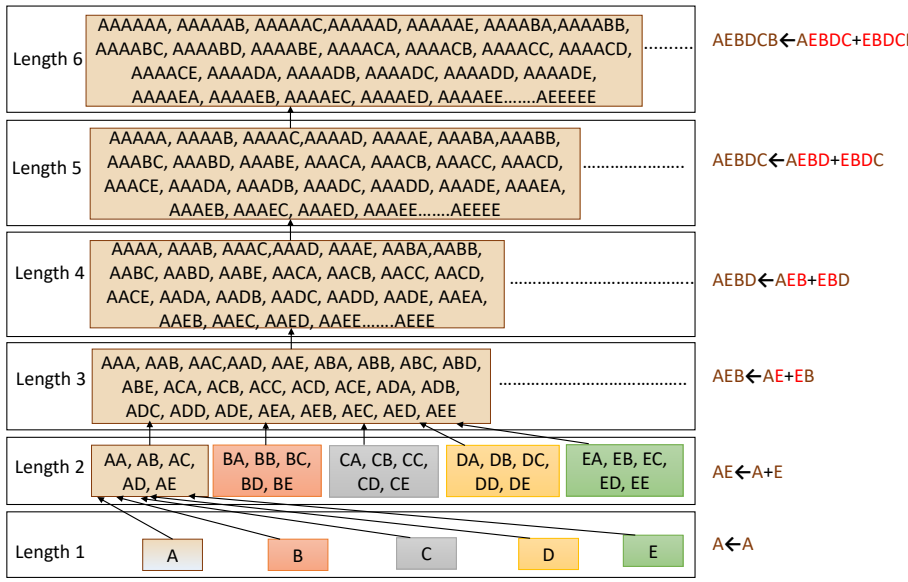


Figure 7. Composition of optimization sequences. Length 2 sequence (AE) is formed by concatenating two length 1 (A + E) sequences. Length 3 (AEB) sequence is formed by merging two sequences of length 2 (AE + EB), only if the first one ends and the second one begins with the same character. Sequences of lengths 4, 5, and 6 are formed similarly.

problem occurs, the previously calculated solutions are used instead of recomputing it, thus computation time is saved [34, 35]. REDUCER retains the unique code sequences by comparing codes with each other. To *estimate the unique code sequences of length 2 to 6*, we have applied Dynamic Programming (DP) technique and compared it with REDUCER. The main motivation behind using DP has been taken from Table 4, where equivalent sequences are a subset of MiCOMP sequences. It implies that large sequences can be constructed by merging two smaller ones. This recursive composition is represented in Figure 7. It can be observed from bottom to top that the length 2 sequence is derived from two length 1 sequences, length 3 from two length 2, and so on. DP initially stores the unique length 1 sequences which derive length 2, then length 2 sequences are stored which derive length 3, and so on.

However, two sequences can be merged only if they have common characters, which means that the second to last characters of the first sequence match with the first to second last characters of the second sequence, as depicted in Figure 7.

The performance of DP is analyzed by means of the sequence estimation accuracy and experiment count increase in Figures 8 and 9. REDUCER shows 100 % accuracy of finding unique sequences for all applications, as it compares each code with the existing ones. Additionally, REDUCER narrows down the experiment

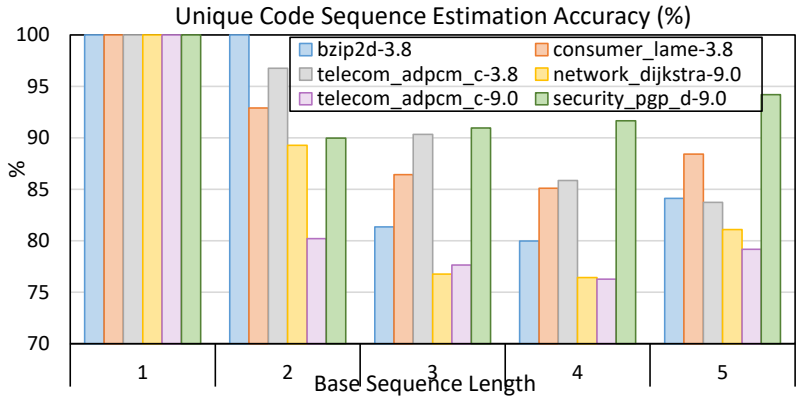


Figure 8. Accuracy of estimating unique optimization sequences via Dynamic Programming (DP) for LLVM-3.8 & LLVM-9.0 (the higher the better). Accuracy computed by dividing correctly estimated DP sequence count with total unique sequence count.

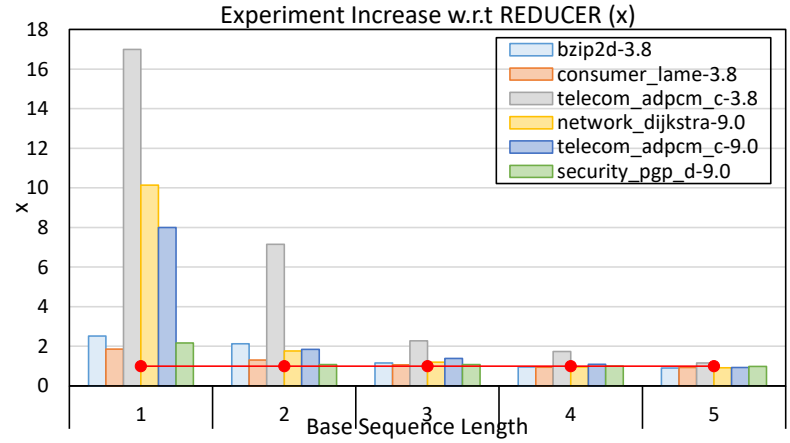


Figure 9. Dynamic Programming (DP) increase in experiment count w.r.t. REDUCER for LLVM-3.8 & LLVM-9.0 (the lower the better). Increase computed by dividing DP estimated experiment count with REDUCER experiment count.

count as it retains only the unique sequences removing all the redundant ones. Unlike REDUCER, DP does not check the actual uniqueness by comparing with existing codes, instead it estimates the unique codes by combining the smaller sequences. This way, DP is likely to be faster than REDUCER. However, with DP there is a higher chance of skipping of unique sequences and inclusion of redundant codes, which results in accuracy loss and experiment count increase.

This work estimates the larger sequences by considering base sequence lengths of 1 to 5. In the case of base length 1, lengths 2 to 6 are estimated by keeping actual

single length unique sequences. For *telecom_adpcm_c_3.8*, *A*, *B*, *C*, and *E* are unique codes. Hence, the larger sequences are formed from these four codes ignoring *D*. In the case of base length 2, length 3 to 6 sequences are estimated by keeping actual double length unique sequences. For *telecom_adpcm_c_3.8*, *AA*, *AB*, *AC*, *AE*, *BA*, *BB*, *BC*, *BE*, *CA*, *CB*, *CE*, *EB*, and *EC* are unique codes. The same criterion is used for base lengths 3, 4, and 5. From Figure 8, 100 % estimation accuracy can be seen for base length 1 in all cases. However, the higher accuracy is at the cost of increased redundant codes. For all applications except *telecom_adpcm_c-3.8*, the single length unique sequences are 5 (A, B, C, D, E), forming 5^2 double length sequences, which results in 5^3, 5^4, 5^5, and 5^6 length 3 to 6 sequences, leading to original 19.5 k experiments. It implies that due to the bottom-up approach, the lower layer behavior is propagated to the upper layers too. As there is no redundancy in the single length sequences, so the upper layers also keep all the sequences. On the contrary, for *telecom_adpcm_c-3.8*, the experiment count is reduced to 5 460, due to four single length unique codes. However, as depicted in Figure 9, still the experiment count is 17× more than REDUCER which is the highest. Similarly, for *network_dijkstra-9.0* experiment count is 10× more.

Furthermore, as per Figure 8, the accuracy starts dropping by increasing the base length, due to missed unique sequences. These misses occur because estimation in base 2, 3, 4, and 5 is done using correct unique sequences. It implies that the problem of estimating larger sequences from smaller ones is not absolute, as the interaction between multiple optimizations is indeterministic. For instance, the sequence *ACCE* might be a unique one, despite its subsequences *ACC* and *CCE* are identical. This behavior can be seen via *bzip2d-3.8*, where accuracy is 100 % for base length 2, but with an increase in length, the accuracy is dropped. It implies that the unique double length combinations are able to derive all the unique higher length sequences, however, the greater length sequences miss several unique combinations. For certain application, a base length shows good accuracy, but for other, the accuracy is not good with the same base. This depends on the interaction effect of optimizations on a certain application, for one application a base covers such sequences whose interaction produces greater unique combinations, thus increasing the accuracy. For other, the effect can be reverse.

The base length increase also reduces the experiment count as the combinations are derived by fixing the unique sequences, which are lesser in quantity. As per Figure 9, for base length 5, all the applications except *telecom_adpcm_c-3.8* show a reduction in experiment count over REDUCER at the cost of lower accuracy. For *security_pgp_d-9.0*, the experiments are lesser than REDUCER at 91.6 % and 94.2 % accuracy for base lengths 4 and 5, respectively. This way, DP is reasonable for *security_pgp_d-9.0* as the experiment increase is not much high for base lengths ≥ 2 and accuracy is decent as well. This behavior is due to the presence of a large number of unique codes in *security_pgp_d-9.0* and also because the base length ≥ 2 sequences are able to derive a higher number of unique sequences of greater lengths. However, the accuracy of less than 100 % makes DP unsuitable because the skipped unique code might be the optimal one with the highest speedup over -O3.

5.4 REDUCER Performance

REDUCER achieves the above experiments reduction (Section 4.3), at the cost of code comparison time which is not present in conventional iterative compilation techniques. In this regard, this section compares the overall time taken by REDUCER and MiCOMP for embedded (Section 5.4.1) and big data (Section 5.4.2) workloads.

5.4.1 Embedded Workloads

A set of embedded applications have been executed from *cBench* suite with dataset *one* on Intel Core i7-8550U laptop machine. The details of the experimental setup and cBench suite are mentioned in Table 1 and 3, respectively. The *cBench* applications have been executed using both REDUCER and MiCOMP for LLVM 3.8, and 9.0. For each application, the execution time is measured by averaging three executions of a loop-wrap. The total time taken by REDUCER is the sum of code *comparison*, *compilation*, and *execution* time. Whereas, MiCOMP total time is the sum of code *compilation* and *execution* time. The individual timings have been reported in Table 5. REDUCER performance is depicted via Figure 10 in terms of total time speedup w.r.t. MiCOMP. Additionally, each application *speedup w.r.t. -O3*, and *optimal sub-sequence* for both LLVM-3.8 and 9.0 have been reported in Table 6. The speedups and optimal sub-sequences are different than reported in [3], due to different test environments.

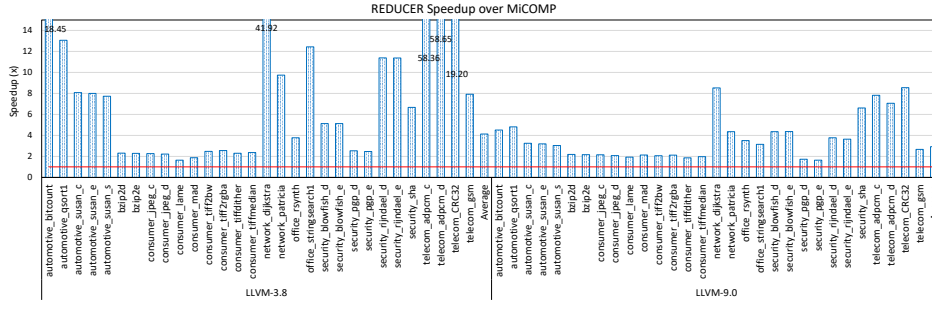


Figure 10. REDUCER time speedup w.r.t. MiCOMP for LLVM-3.8 & LLVM-9.0

As per Table 5, for all the applications the IR comparison time is significantly smaller than the compilation and execution time. This way, REDUCER's prior IR comparison cuts down the experiment count without increasing the time and resource overheads. Hence, the compilation and execution times are shorter for all applications, because these have been measured only for the unique codes which are lesser than the original 19.5 k. Further, a larger comparison time can be seen for applications possessing a greater number of unique codes, due to an increased number of comparisons. Also, the comparison time is increased for larger code

Applications	LLVM-3.8		LLVM-9.0	
	REDUCER Comparison + Compilation & Execution Time	**MiCOMP Execution Time**	**REDUCER Comparison + Compilation & Execution Time**	**MiCOMP Execution Time**
automotive_bitcount	2 m 7.437 s + 1 h 45 m 45.072 s	33 h 9 m 48.835 s	8 m 12.073 s + 12 h 32 m 36.607 s	57 h 18 m 24.959 s
hline automotive_qsort1	2 m 3.219 s + 4 h 6 m 26.163 s	54 h 5 m 32.677 s	13 m 17.055 s + 11 h 48 m 30.306 s	57 h 55 m 5.473 s
hline automotive_susan_c	19 m 15.745 s + 37 h 32 m 1.081 s	305 h 42 m 1.198 s	1 h 27 m 54.887 s + 97 h 31 m 36.153 s	321 h 48 m 36.468 s
hline automotive_susan_e	19 m 0.523 s + 17 h 22 m 1.422 s	141 h 26 m 58.470 s	1 h 28 m 18.939 s + 45 h 51 m 51.6 s	151 h 20 m 20.859 s
hline automotive_susan_s	19 m 0.354 s + 5 h 58 m 43.483 s	48 h 41 m 42.396 s	1 h 27 m 39.483 s + 17 h 11 m 42.905 s	56 h 44 m 21.810 s
hline bzip2d	2 h 25 m 29.997 s + 26 h 36 m 1.811 s	67 h 6 m 52.703 s	4 h 18 m 40.583 s + 24 h 16 m 6.252 s	62 h 38 m 18.913 s
hline bzip2e	2 h 27 m 48.848 s + 24 h 25 m 43.140 s	61 h 38 m 5.735 s	4 h 18 m 19.182 s + 22 h 34 m 24.166 s	58 h 15 m 48.966 s
hline consumer_jpeg_c	4 h 28 m 5.858 s + 34 h 58 m 47.546 s	89 h 40 m 10.199 s	6 h 52 m 26.987 s + 32 h 34 m 40.058 s	85 h 5 m 52.390 s
hline consumer_jpeg_d	4 h 6 m 43.359 s + 25 h 51 m 7.058 s	66 h 31 m 56.764 s	6 h 42 m 40.772 s + 25 h 48 m 32.593 s	67 h 24 m 28.608 s
hline consumer_lame	4 h 40 m 17.279 s + 35 h 50 m 39.655 s	66 h 39 m 17.488 s	6 h 5 m 45.341 s + 28 h 3 m 20.393 s	66 h 3 m 30.571 s
hline consumer_mad	3 h 16 m 23.984 s + 73 h 30 m 15.024 s	143 h 41 m 25.161 s	6 h 57 m 39.909 s + 54 h 12 m 10.991 s	130 h 42 m 44.686 s
hline consumer_tiff2bw	3 h 54 m 23.318 s + 28 h 32 m 20.286 s	80 h 23 m 50.338 s	6 h 49 m 58.180 s + 32 h 12 m 45.765 s	80 h 47 m 40.070 s
hline consumer_tiff2rgba	3 h 53 m 25.429 s + 39 h 4 m 29.516 s	110 h 4 m 41.021 s	6 h 48 m 40.286 s + 47 h 25 m 19.920 s	115 h 24 m 52.931 s
hline consumer_tiffdither	3 h 51 m 34.296 s + 17 h 29 m 27.2 s	49 h 11 m 18.818 s	6 h 46 m 21.959 s + 19 h 37 m 32.768 s	49 h 13 m 28.066 s
hline consumer_tiffmedian	4 h 19 m 45.241 s + 22 h 53 m 27.874 s	64 h 19 m 43.911 s	7 h 17 m 12.163 s + 27 h 2 m 49.544 s	67 h 49 m 46.514 s
hline network_dijkstra	1 m 18.064 s + 6 m 53.783 s	5 h 43 m 36.315 s	7 m 7.499 s + 38 m 15.240 s	6 h 26 m 55.092 s
hline network_patricia	1 m 54.758 s + 3 h 9 m 13.651 s	31 h 2 m 48.228 s	9 m 22.514 s + 8 h 20 m 48.717 s	37 h 2 m 31.854 s
hline office_rsynth	43 m 53.006 s + 10 h 52 m 36.417 s	43 h 50 m 13.451 s	1 h 10 m 37.906 s + 13 h 47 m 16.173 s	52 h 32 m 39.824 s
hline office_stringsearch1	3 m 17.225 s + 3 h 9 m 22.507 s	39 h 57 m 4.458 s	15 m 19.732 s + 4 h 53 m 5.920 s	16 h 12 m 43.740 s
hline security_blowfish_d	14 m 17.719 s + 16 h 6 m 48.013 s	83 h 55 m 21.205 s	22 m 50.546 s + 17 h 1 m 40.431 s	75 h 52 m 39.645 s
hline security_blowfish_e	14 m 10.497 s + 16 h 23 m 46.093 s	85 h 23 m 43.636 s	22 m 43.436 s + 18 h 4 m 13.958 s	80 h 31 m 25.664 s
hline security_pgp_d	4 h 11 m 42.643 s + 19 h 33 m 58.447 s	60 h 8 m 0.460 s	7 h 2 m 54.529 s + 27 h 14 m 40.322 s	59 h 11 m 45.659 s
hline security_pgp_e	4 h 13 m 47.843 s + 17 h 6 m 0.036 s	52 h 33 m 14.188 s	7 h 1 m 54.334 s + 22 h 1 m 1.559 s	47 h 50 m 16.878 s
hline security_rijndael_d	7 m 40.857 s + 5 h 13 m 48.272 s	60 h 59 m 3.278 s	37 m 33.950 s + 14 h 28 m 10.230 s	56 h 59 m 58.524 s
hline security_rijndael_e	7 m 39.601 s + 5 h 9 m 9.975 s	60 h 4 m 58.253 s	37 m 31.819 s + 7 h 44 m 50.671 s	30 h 31 m 9.446 s
hline security_sha	6 m 38.600 s + 16 h 3 m 59.443 s	107 h 50 m 0.057 s	10 m 8.242 s + 13 h 45 m 6.575 s	92 h 4 m 35.121 s
hline telecom_adpcm_c	1 m 16.927 s + 54 m 50.166 s	54 h 35 m 14.502 s	7 m 16.430 s + 6 h 7 m 19.345 s	48 h 50 m 37.601 s
hline telecom_adpcm_d	1 m 16.785 s + 1 h 9 m 40.124 s	69 h 21 m 9.758 s	7 m 9.597 s + 7 h 58 m 36.192 s	57 h 9 m 3.196 s
hline telecom_CRC32	1 m 6.076 s + 1 h 23 m 20.919 s	27 h 1 m 23.813 s	6 m 21.450 s + 3 h 10 m 10.373 s	27 h 59 m 8.641 s
hline telecom_gsm	16 m 42.200 s + 8 h 22 m 53.616 s	68 h 36 m 31.154 s	1 h 22 m 6.525 s + 20 h 30 m 31.041 s	58 h 26 m 57.930 s
hline **Mean**	**1 h 37 m 44.256 s + 17 h 21 m 27.260 s**	**74 h 26 m 51.616 s**	**2 h 54 m 48.210 s + 22 h 48 m 59.559 s**	**72 h 32 m 31.670 s**

Table 5. Comparison of REDUCER with MiCOMP

sizes. For instance, *telecom_adpcm_c* (LLVM-3.8) and *network_dijkstra* (LLVM-9.0) take lesser time because the identical codes are already generated by some previous sub-sequence, stopping the comparison for a code the moment its identical is found. It can be observed that the comparison time of *telecom_CRC32* (LLVM-3.8 & LLVM 9.0) is slightly lesser than *telecom_adpcm_c* (LLVM-3.8) and *network_dijkstra* (LLVM-9.0) due to the smaller IR code size of *telecom_CRC32*. The redundancy proportion of *telecom_CRC32* is higher but lesser than the maximum identical codes count of *telecom_adpcm_c* (LLVM-3.8) and *network_dijkstra* (LLVM-9.0). On the contrary, due to comparing each code with a large number of unique codes, the *consumer_lame* (LLVM-3.8) and *security_pgp_d* (LLVM-9.0) exhibit the maximum comparison time.

Applications	LLVM-3.8		LLVM-9.0		Speedup
	Speedup w.r.t. -O3	Optimal Sub-Sequence	Speedup w.r.t. -O3	Optimal Sub-Sequence	Optimal LLVM 9.0 w.r.t. 3.8
automotive_bitcount	1.08×	ACACA	1.21×	BDADB	1.0×
automotive_qsort1	1.08×	AABCAB	1.04×	CAAADB	0.90×
automotive_susan_c	1.37×	AAAAAE	1.08×	AAAAAA	1.03×
automotive_susan_e	1.23×	BCCEEB	1.06×	AAABCB	1.26×
automotive_susan_s	1.07×	AAAACA	1.24×	AAAAAD	1.008×
bzip2d	1.40×	BDECED	1.14×	DADAC	1.03×
bzip2e	1.46×	BBDE	1.08×	CDBAAB	1.03×
consumer_jpeg_c	1.17×	BAD	1.56×	ACDCDA	1.35×
consumer_jpeg_d	1.45×	BADAAE	1.02×	AAAAAD	1.06×
consumer_lame	1.01×	DBECAA	1.25×	AADDBC	1.28×
consumer_mad	1.23×	ABAC	1.11×	AACBCD	1.01×
consumer_tiff2bw	1.16×	BCEABC	1.05×	BECCCD	1.01×
consumer_tiff2rgba	1.01×	ABAEDA	1.24×	AAAABD	1.43×
consumer_tiffdither	1.08×	EABECB	1.06×	BCDAA	1.01×
consumer_tiffmedian	1.27×	CAEAEB	1.28×	CBCABA	1.02×
network_dijkstra	1.19×	AAAA	1.09×	AAAAB	0.87×
network_patricia	1.13×	AAAABC	1.06×	AAAABC	0.81×
office_rsynth	1.05×	AAAABC	1.26×	BCDCCB	0.98×
office_stringsearch1	1.08×	CABAAE	1.12×	BACCE	1.06×
security_blowfish_d	1.11×	ABADEE	1.006×	BCACAD	1.05×
security_blowfish_e	1.08×	EBAE	1.006×	BBCACA	1.01×
security_pgp_d	1.17×	ADAACC	1.05×	DDABC	1.23×
security_pgp_e	1.05×	BCACDE	1.07×	BDDCBA	1.07×
security_rijndael_d	1.11×	BECACA	1.18×	AACAAC	1.19×
security_rijndael_e	1.12×	ABECAB	1.24×	BBDCBC	1.19×
security_sha	1.13×	CBACAC	1.06×	BABDCC	1.01×
telecom_adpcm_c	1.48×	BB	1.79×	B	1.33×
telecom_adpcm_d	1.15×	CAE	1.24×	CCACDB	1.24×
telecom_CRC32	1.08×	AAAABC	1.06×	BCABD	1.004×
telecom_gsm	1.31×	EBBAAB	1.03×	DCCAAB	1.02×
Mean	**1.16**×	–	**1.14**×	–	**1.06**×

Table 6. Optimal speedup for LLVM 3.8 and 9.0

Despite increased comparison time, REDUCER clearly outperforms MiCOMP for LLVM-3.8 and 9.0, which can be observed via Table 5. For all the applications, REDUCER takes lesser time, as compared to MiCOMP. As depicted via Figure 10, for LLVM-3.8, the maximum speedup of 58.6× is observed for *telecom_adpcm_d*, because a large number of repeated executions have been suppressed. The lowest speedup observed is 1.64× for *consumer_lame*, due to less number of repeated codes.

On average, for LLVM-3.8, a decent speedup of 4.13× is seen. Similarly, LLVM-9.0 shows a maximum speedup of 8.54× for *telecom_CRC32*, lowest speedup of 1.64× for *security_pgp_e*, and average speedup of 2.92×.

The redundancy count is not the only parameter to affect REDUCER performance, instead it is equally affected by code size. For LLVM-9.0, *network_dijkstra* depicts the highest redundancy count, but its speedup is not dominating because its comparison time is greater due to code size. Conversely, *telecom_CRC32* shows the highest speedup due to both smaller code size than *network_dijkstra* and higher redundancy count than the majority of other applications. In this manner, for longer sequence lengths (> 6) and smaller code sizes, the performance of REDUCER is expected to increase exponentially w.r.t. MiCOMP, as Figure 6 depicts the substantial increase in redundancy proportion for longer sequence lengths.

Despite same workloads, the MiCOMP and REDUCER speed is different for both LLVM-3.8 and 9.0, which is possibly due to wide differences in the compiler versions resulting in varying -O3 internal passes. This way, the constructed subsequences widely vary for both versions. It can be observed via Table 5, despite a same number of experiments, the MiCOMP (LLVM-9.0) average speed is higher than LLVM-3.8 because, for the majority of applications, LLVM-9.0 generated codes are executed in lesser time than LLVM-3.8 due to enhanced optimizations. On the contrary, REDUCER (LLVM-3.8) is on the average 1.35× faster than REDUCER (LLVM-9.0). Primarily, two factors are affecting the speed of REDUCER, i.e. experiment count and code execution time. For instance, REDUCER (LLVM-9.0) is slower for *telecom_adpcm_c*, because it is required to process 2.4 k codes which are only 327 with LLVM-3.8. Conversely, for a few applications REDUCER (LLVM-9.0) dominates REDUCER (LLVM-3.8) but with a minor margin, for instance in *bzip2d*, the processing is reduced to 7.5 k, which is 7.7 k with LLVM-3.8. The redundancy count is varied by the impact caused by optimization sequences on a given application, which is indeterministic.

Overall, the sub-sequences are able to exploit reasonable speedup for majority applications as evident by Table 6. On average speedup[9] of 1.16× and 1.14×, are achieved for LLVM-3.8 and LLVM-9.0, respectively. Besides, the maximum speedup is 1.48× and 1.79× for LLVM-3.8 and LLVM-9.0 *telecom_adpcm_c*, respectively. With LLVM-9.0, -O3 has become even more powerful due to increased optimization passes, thus it gets tougher to beat -O3 performance. This way, the speedup w.r.t. -O3 is observed to be lesser for LLVM-9.0.

As per Table 6, LLVM-9.0 optimal sub-sequence is showing greater speedup w.r.t. LLVM-3.8 optimal sub-sequence. The maximum speedup of 1.43× is seen for *consumer_tiff2rgba*, and the average speedup is 1.06×. The speedup is possibly due to LLVM-9.0 being faster than LLVM-3.8 with an enhanced set of optimization passes. Besides, the speedup is greater due to our efficient implementation of MiCOMP for LLVM-9.0, which means that we have derived better LLVM-9.0 -O3 sub-sequences using k-means than the ones reported in [3].

[9] Harmonic mean is used to average the speedup gains [3].

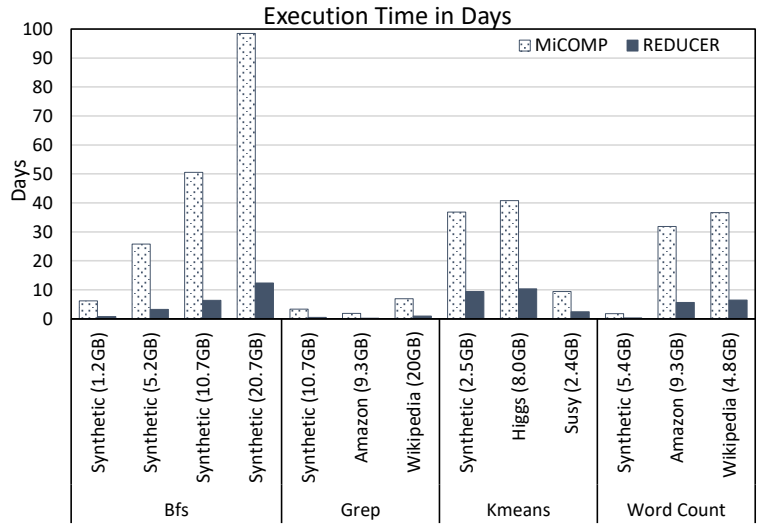
5.4.2 Big Data Workloads

Several well known C/C++ based applications from Rodinia [28], Phoenix [31], CortexSuite [30], genann [32], and grep-bench [29] benchmarks have been tested. Only those applications have been selected which are part of standard big data benchmarks, representing graph mining, classification, clustering, and statistics categories. These include bfs, grep, k-means, word count, etc., as discussed in Table 7. These benchmarks have been run on an Intel Xeon Server machine whose details are listed in Table 1. Each application has been run with 3 to 4 datasets of varying sizes and formats.

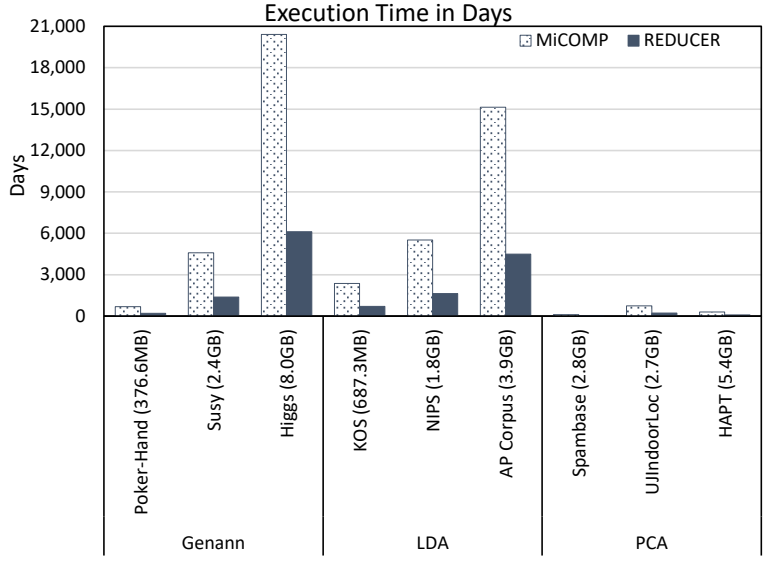
Application	Description	Input Dataset Format
Breadth-First Search (BFS) [28]	Traverses a graph in a breadthward motion.	Graph generated by specifying the number of nodes.
Grep [29]	Searches a file for a particular pattern of characters, and displays the lines containing that pattern.	Text file containing words.
k-means [28]	Represents the data objects by the centroids of the sub-clusters by dividing a cluster of data objects into k sub-clusters.	Dataset consisted of a set of numeric features.
Word Count (WC) [31]	Counts the frequency of occurrence of each unique word in a text document.	Text files containing words.
Gennan [32]	Neural network library for using and training feedforward artificial neural networks (ANN).	Numeric predictive attributes and the class.
Latent Dirichlet Allocation (LDA) [30]	Topic modeling algorithm that is used in natural language processing for discovering topics from unordered documents.	Document is represented as a sparse vector of word counts, in the form: [M][term_1]:[count]... [term_N]:[count]
Principle Component Analysis (PCA) [30]	It is a statistical technique for feature extraction in multivariate datasets.	Data numeric attributes and the class.

Table 7. Benchmark details

The impact of REDUCER is more prominent with big data workloads which consumes larger execution time than conventional cases. As can be observed via Figure 11, for all the applications and datasets REDUCER makes the experimentation feasible by bringing a substantial reduction in execution time. For each application, the comparison and compilation are done only one time. This way, in the Figure 11, only the bar corresponding to the first dataset includes the comparison and compilation times along with execution time, the other bars only involve



a)



b)

Figure 11. REDUCER performance for big data applications

Technique	Environment	Benchmark Suite	Experiment Count per Application	Avg Exe Time Imp w.r.t. Baseline	Collection Time
Less is More [14]	Cortex-M0, LLVM-3.8 Cortex-M3, LLVM-5.0	BEEBS	50, 64 optimizations + (O2 baseline)	2.4 % 5.3 %	–
Lost in translation [15]	Intel i5-6300U, LLVM-6.0 Arm Cortex-A53, LLVM-6.0	CK Milepost-GCC-Codelet	66, 64 optimizations +(O3 baseline)	11.5 % 5.1 %	–
IODC [12]	Intel Xeon, AMD, & Loongson clusters, GCC-4.4	MapReduce & Server Applications	300 random optimizations +(O3 baseline)	32.43 % 10.71 %	110 days 740 days
FFD [27]	Cortex-M0, Cortex-M3, Cortex-A8, Epiphany, XMOS L1, GCC-4.7	MiBench & WCET Applications	2 048 optimizations +(O1, O2 baseline)	–	–
Sensitivity Analysis [10]	Intel Core i7 LLVM-3.8.1	BEEBS	1 728 000	–	–
Hybrid Approach [11]	Intel Core i7-3779, LLVM 3.5	Polybench & cBench	– +(O3 baseline)	8.01 % 6.07 %	–
MiCOMP [3]	Intel Xeon LLVM-3.8 Intel Core i7-8550U, LLVM-3.8, LLVM-9.0	Ctuning cBench	19 530 optimizations +(O3 baseline)	16.66 % 14.41 % 12.50 %	 93 days 91 days
REDUCER	Intel Core i7-8550U, LLVM-3.8, LLVM-9.0	Ctuning cBench	Avg reduction w.r.t. MiCOMP **77.60 %**, **69.98 %**	14.41 % 12.50 %	**24 days** **33 days**

Table 8. REDUCER comparison with existing works

execution time. Hence, the comparison time is spent only for the first dataset execution, the rest datasets execution is comparison free. REDUCER removes the redundant codes at the start, hence executes all the datasets with a reduced number of codes. Whereas, in MiCOMP, each dataset is executed with all the codes including both the unique and identical. It can be observed via Figure 11, REDUCER greatly facilitates the iterative compilation of *bfs (Synthetic-20.7 GB)* dataset, by cutting down execution time to only 12 days from 98 days of MiCOMP. Similarly, for iterative compilation of *PCA (Spambase-2.8 GB)* dataset, only 32 days are required with REDUCER in comparison to 111 days of MiCOMP. Similarly, for other workloads, it can be seen how REDUCER makes the iterative compilation feasible for big data workloads comprising of high volume and variety datasets.

6 RELATED WORK

Several works [13, 22, 21, 20, 19] have emphasized on code comparisons for removing redundant executions. [22, 21, 20, 19] were based on VPO (Very Portable Optimizer) compiler back end, which performed all the analyses and optimizations on a single low-level representation called Register Transfer Lists (RTLs). It detected the iden-

tical function instances by performing three checks including instructions count, instructions byte-sum, and the CRC (Cyclic Redundancy Code) checksum on the bytes of the RTLs. Similarly, in [13] MD5 checksum of assembler code was obtained to verify that no two optimizations combinations generate the same binary. The work selected GCC 200 optimization combinations using a random search strategy. In comparison to these, REDUCER to the best of our knowledge is the first work that detects identical codes in LLVM by comparing the complete IR codes. The granularity of comparison is complete IR code, not just a function or basic block instance.

Conversely, other works [14, 15, 3, 12, 10, 27, 11] did not make the code level comparisons and executed the same code repeatedly. The comparison of these works with REDUCER is depicted via Table 8. In [14, 15], initially, the required performance metrics were tested using standard optimization levels (O2, O3, etc). Then, the metrics were measured by excluding one pass at a time from the standard optimization level, till all the passes were eliminated. In this way, the tested configuration count was lesser, because only the passes present in the standard LLVM optimization level were considered. With this approach, the search space is reduced, but the performance improvement is significantly lesser than the other approaches.

In [12], iterative optimization for the data center (IODC) was proposed which found the optimal compiler configurations for Map Reduce and server applications involving a large collection of massive size datasets. IODC showed greater speedup but at the cost of collection time of 850 days. Despite testing only 300 randomly chosen combinations of compiler optimizations, the collection time was higher due to the execution of a single application with multiple large datasets. However, if the redundancy fraction is $f\,\%$ in the derived optimizations, then all the datasets are required to be executed with these redundant $f\,\%$ codes, increasing the time significantly. In this situation, the integration of REDUCER with IODC can achieve the reported speedup in lesser runs, with a substantial reduction in data collection time, because for an application the redundant codes are checked only one time irrespective of the number of datasets.

The authors in [27] proposed fractional factorial design which reduced the search space for finding optimal optimization combination in GCC. [10] performed sensitivity test for analyzing the impact of 54 LLVM code optimizations on the execution time of applications. Similarly, a design-space exploration was proposed in [11] for searching compiler optimization sequence. The given hybrid approach found optimizations and their order of application, through previously generated sequences for training programs set. Initially, a clustering algorithm selected optimizations, followed by a metaheuristic algorithm for discovering the sequence of optimizations. As per results, the discovered optimized code sequences on average brought the only improvement of 8.01 % and 6.07 % w.r.t. -O3, which is less in comparison to other works. In [3] MiCOMP was proposed, which reduced the search space from billions to few thousand. It did so by clustering LLVM -O3 optimizations into five sub-sequences by using agglomerative clustering. The search space was reduced

because phase ordering was exploited using sub-sequences, instead of individual optimizations. Overall MiCOMP showed significant performance improvement relative to -O3, with 5 clusters and a maximum sequence length of 6 (total 19.5 k experiments). By comparing the proposed works listed in Table 8 with MiCOMP, it can be observed that MiCOMP searches the optimal optimization sequence (better than -O3) in lesser runs for an application. Also, it is evident that MiCOMP sub-sequences exploit greater speedup (w.r.t. -O3) than others. However, MiCOMP did not exclude the identical codes present in 19.5 k sub-sequences, instead, all the permutations were executed to find optimal sub-sequences, increasing the data collection time.

This paper reduces MiCOMP search space by proposing REDUCER which is responsible for eliminating identical codes. In this manner, the repeated execution of the same code is prevented, saving the testing time without affecting the performance accuracy. As per Table 8, the performance improvements and data collection time of MiCOMP reported in [3] and MiCOMP implemented in our work are not comparable because each technique has been tested on different test environments. It can be observed that REDUCER shortens the data collection time to 24 and 33 days (LLVM-3.8 & LLVM-9.0) from MiCOMP's 93 and 91 days without sacrificing the performance improvement w.r.t. baseline. Further, we have extended MiCOMP for LLVM-9.0 by constructing 5 clusters using k-means clustering. Our derived optimization sub-sequences shows average speedup of 1.06× w.r.t. MiCOMP (LLVM-3.8) sub-sequences given in [3].

7 CONCLUSION

The compiler search space reduction technique REDUCER has been presented in this paper. REDUCER relies on straightforward code comparisons to inhibit *identical* code executions. REDUCER has been tested using well-accepted MiCOMP iterative compilation technique with LLVM-3.8 and 9.0. As per reported results, REDUCER substantially accelerates the iterative compilation process in comparison to MiCOMP by eliminating a large number of redundant experiments. In this regard, REDUCER completes the overall iterative compilation of embedded workloads within 24 and 33 days (LLVM-3.8 & LLVM-9.0), respectively, whereas MiCOMP takes 93 and 91 days for the same task.

The promising results of REDUCER (LLVM-9.0) anticipate the high significance of REDUCER for forthcoming compilers as well. Furthermore, REDUCER is proved to be significantly faster for *big data* workloads. Besides, it is found to be simple, generic, and easily adaptable in any iterative compilation technique. In the future, we intend to reduce comparison time by implementing a parallel version of REDUCER. Presently, REDUCER can only detect identical codes, in the future REDUCER will be extended to detect equivalent codes as well, which is expected to further reduce the search space.

REFERENCES

[1] LLVM 2019 (Accessed October 20, 2019). The LLVM Compiler Infrastructure. `https://llvm.org/`.

[2] Aho, A.: Compilers: Principles, Techniques, and Tools (for Anna University). 2/e, 2003.

[3] Ashouri, A. H.—Bignoli, A.—Palermo, G.—Silvano, C.—Kulkarni, S.—Cavazos, J.: MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 14, 2017, No. 3, Art. No. 29, pp. 1–28, doi: 10.1145/3124452.

[4] Ashouri, A. H.—Killian, W.—Cavazos, J.—Palermo, G.—Silvano, C.: A Survey on Compiler Autotuning Using Machine Learning. ACM Computing Surveys (CSUR), Vol. 51, 2019, No. 5, Art. No. 96, pp. 1–42, doi: 10.1145/3197978.

[5] Ashouri, A. H.—Mariani, G.—Palermo, G.—Park, E.—Cavazos, J.—Silvano, C.: COBAYN: Compiler Autotuning Framework Using Bayesian Networks. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 13, 2016, No. 2, Art. No. 21, pp. 1–25, doi: 10.1145/2928270.

[6] Bodin, F.—Kisuki, T.—Knijnenburg, P.—O'Boyle, M.—Rohou, E.: Iterative Compilation in a Non-Linear Optimisation Space. Workshop on Profile and Feedback-Directed Compilation, 1998.

[7] Chen, M.—Mao, S.—Liu, Y.: Big Data: A Survey. Mobile Networks and Applications, Vol. 19, 2014, No. 2, pp. 171–209, doi: 10.1007/s11036-013-0489-0.

[8] Chong, F. T.—Franklin, D.—Martonosi, M.: Programming Languages and Compiler Design for Realistic Quantum Hardware. Nature, Vol. 549, 2017, No. 7671, pp. 180–187, doi: 10.1038/nature23459.

[9] Cooper, K.—Torczon, L.: Engineering a Compiler. Elsevier, 2011.

[10] de la Torre, J. C.—Ruiz, P.—Dorronsoro, B.—Galindo, P. L.: Analyzing the Influence of LLVM Code Optimization Passes on Software Performance. In: Medina, J., Ojeda-Aciego, M., Verdegay, J., Perfilieva, I., Bouchon-Meunier, B., Yager, R. (Eds.): Information Processing and Management of Uncertainty in Knowledge-Based Systems. Applications (IPMU 2018). Springer, Cham, Communications in Computer and Information Science, Vol. 855, 2018, pp. 272–283, doi: 10.1007/978-3-319-91479-4_23.

[11] de Souza Xavier, T. C.—da Silva, A. F.: Exploration of Compiler Optimization Sequences Using a Hybrid Approach. Computing and Informatics, Vol. 37, 2018, No. 1, pp. 165–185, doi: 10.4149/cai_2018_1_165.

[12] Fang, S.—Xu, W.—Chen, Y.—Eeckhout, L.—Temam, O.—Chen, Y.—Wu, C.—Feng, X.: Practical Iterative Optimization for the Data Center. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 12, 2015, No. 2, Art. No. 15, pp. 1–26, doi: 10.1145/2739048.

[13] Fursin, G.—Temam, O.: Collective Optimization: A Practical Collaborative Approach. ACM Transactions on Architecture and Code Optimization (TACO) Vol. 7, 2010, No. 4, Art. No. 20, pp. 1–29, doi: 10.1145/1880043.1880047.

[14] Georgiou, K.—Blackmore, C.—Xavier-de-Souza, S.—Eder, K.: Less Is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption. 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES '18), 2018, pp. 35–42, doi: 10.1145/3207719.3207727.

[15] Georgiou, K.—Chamski, Z.—Amaya Garcia, A.—May, D.—Eder, K.: Lost in Translation: Exposing Hidden Compiler Optimization Opportunities. The Computer Journal, 2020, doi: 10.1093/comjnl/bxaa103.

[16] Guthaus, M. R.—Ringenberg, J. S.—Ernst, D.—Austin, T. M.—Mudge, T.—Brown, R. B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. Fourth Annual IEEE International Workshop on Workload Characterization (WWC-4), 2001, pp. 3–14, doi: 10.1109/WWC.2001.990739.

[17] Hall, M.—Padua, D.—Pingali, K.: Compiler Research: The Next 50 Years. Communications of the ACM, Vol. 52, 2009, No. 2, pp. 60–67, doi: 10.1145/1461928.1461946.

[18] Hoste, K.—Eeckhout, L.: Cole: Compiler Optimization Level Exploration. 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08), 2008, pp. 165–174, doi: 10.1145/1356058.1356080.

[19] Jantz, M. R.—Kulkarni, P. A.: Analyzing and Addressing False Interactions During Compiler Optimization Phase Ordering. Software: Practice and Experience, Vol. 44, 2014, No. 6, pp. 643–679, doi: 10.1002/spe.2176.

[20] Kulkarni, P.—Hines, S.—Hiser, J.—Whalley, D.—Davidson, J.—Jones, D.: Fast Searches for Effective Optimization Phase Sequences. ACM SIGPLAN Notices, Vol. 39, 2004, No. 6, pp. 171–182, doi: 10.1145/996893.996863.

[21] Kulkarni, P. A.—Whalley, D. B.—Tyson, G. S.—Davidson, J. W.: Exhaustive Optimization Phase Order Space Exploration. International Symposium on Code Generation and Optimization (CGO '06), 2006, pp. 1–13, doi: 10.1109/CGO.2006.15.

[22] Kulkarni, P. A.—Whalley, D. B.—Tyson, G. S.—Davidson, J. W.: Practical Exhaustive Optimization Phase Order Exploration and Evaluation. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 6, 2009, No. 1, Art. No. 1, pp. 1–36, doi: 10.1145/1509864.1509865.

[23] Lopes, B. C.—Auler, R.: Getting Started with LLVM Core Libraries. Packt Publishing Ltd, 2014.

[24] Sarda, S.—Pandey, M.: LLVM Cookbook. Packt Publishing Ltd, 2015.

[25] Triantafyllis, S.—Vachharajani, M.—Vachharajani, N.—August, D. I.: Compiler Optimization-Space Exploration. International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO 2003), 2003, pp. 204–215, doi: 10.1109/CGO.2003.1191546.

[26] Zhang, W.—Zhao, D.—Wang, X.: Agglomerative Clustering via Maximum Incremental Path Integral. Pattern Recognition, Vol. 46, 2013, No. 11, pp. 3056–3065, doi: 10.1016/j.patcog.2013.04.013.

[27] Pallister, J.—Hollis, S. J.—Bennett, J.: Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms. The Computer Journal, Vol. 58, 2015, No. 1, pp. 95–109, doi: 10.1093/comjnl/bxt129.

[28] Che, S.—Boyer, M.—Meng, J.—Tarjan, D.—Sheaffer, J. W.—Lee, S. H.—Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 44–54, doi: 10.1109/IISWC.2009.5306797.

[29] Grep-Bench, 2019 (Accessed March 18, 2019). `https://github.com/pokle/grep-bench`.

[30] Thomas, S.—Gohkale, C.—Tanuwidjaja, E.—Chong, T.—Lau, D.—Garcia, S.—Taylor, M. B.: CortexSuite: A Synthetic Brain Benchmark Suite. IEEE International Symposium on Workload Characterization (IISWC), 2014, pp. 76–79, doi: 10.1109/IISWC.2014.6983043.

[31] Yoo, R. M.—Romano, A.—Kozyrakis, C.: Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 198–207, doi: 10.1109/IISWC.2009.5306783.

[32] C Neural Network Library: Genann, 2019 (Accessed March 01, 2019). `https://codeplea.com/genann`.

[33] Purini, S.—Jain, L.: Finding Good Optimization Sequences Covering Program Space. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 9, 2013, No. 4, Art. No. 56, pp. 1–23, doi: 10.1145/2400682.2400715.

[34] Nalbantoğlu, Ö. U.: Dynamic Programming. In: Russell, D. (Ed.): Multiple Sequence Alignment Methods. Humana Press, Totowa, NJ, Methods in Molecular Biology (Methods and Protocols), Vol. 1079, 2014, pp. 3–27, doi: 10.1007/978-1-62703-646-7_1.

[35] Dasgupta, S.—Papadimitriou, C. H.—Vazirani, U. V.: Dynamic Programming. In: Dasgupta, S., Papadimitriou, C. H., Vazirani, U. V. (Eds.): Algorithms. Chapter 6. Vol. 1, 2006, pp. 169–199.

[36] Han, J.—Pei, J.—Kamber, M.: Data Mining Concepts and Techniques. The Morgan Kaufmann Series in Data Management Systems, 2011, pp. 83–124.

[37] Steinbach, M.—Kumar, V.—Tan, P. N.: Cluster Analysis: Basic Concepts and Algorithms. Introduction to Data Mining, Pearson Addison Wesley, 2005.

Hameeza Ahmed received her M.Eng. and B.Eng. degrees in computer and information systems from the NED University of Engineering and Technology, Pakistan in 2015 and 2012, respectively. She is currently pursuing her Ph.D. from the same university. Her research interests include big data computing, compiler optimizations, and computer architecture.

Muhammad Ali Ismail is Professor and Chair at the Department of Computer and Information Systems Engineering, NED University of Engineering and Technology. He is also serving as Director of the High Performance Computing Center and Scientific Director of the Exascale Open Data Analytics Lab, National Center in Big Data and Cloud Computing at the same university. He has more than 16 years experience of research, teaching and administration in both national and international universities. He received his Ph.D. in high performance computing in 2011. Afterwards he pursued his post doctorate in automatic design space exploration from ULBS Romania and become a HiPEAC member. He has published over 65 scientific papers in international journals and conferences along with a U.S. patent. He has won many of the national and international grants of worth above Rs. 200 Million. He is also the recipient of Research Productivity Award by Pakistan Council for Science and Technology, Ministry of Science and Technology, Government of Pakistan. His current research interests include computational HPC, big data mining, cluster and cloud computing, multicore processor architecture and programming, machine learning, heuristics and automatic design space exploration. He is also serving IET Karachi Network as its Vice Chairman.