Speeding Up FMSA with Linear Time Grouping

Liu Jiang
EECS Department
University of Michigan
Ann Arbor, USA
ljlovecs@umich.edu

Leiqi Ye
EECS Department
University of Michigan
Ann Arbor, USA
yeleiqi@umich.edu

Ying Yuan
EECS Department
University of Michigan
Ann Arbor, USA
yyyuan@umich.edu

Franklin Kong
EECS Department
University of Michigan
Ann Arbor, USA
ftk@umich.edu

***Abstract*—This project focuses on reducing compilation time on Function Merging by Sequence Alignment (FMSA). In finding candidate functions to merge, FMSA suffers from quadratic pairwise comparison of all functions within the compilation unit. We propose to use reference vector and grouping to first reduce the scope of comparison. Secondly, we change the comparison metrics from Manhattan Distance to Cosine Similarity. Finally, we skip unprofitable and time-consuming small-sized functions. This implementation achieves 21% compilation speedup compared to the best configuration of original FMSA with only around 0.5% loss in code size reduction.** [1]

I. Introduction

In this section, we will discuss and introduce the background and the implementation of Function Merging by Sequence Alignment (FMSA), as well as the problem that currently exists in FMSA.

A. Background

In recent years, resource-constrained devices have become increasingly important. Application binaries for these devices often reach several megabytes in size, turning memory size into a limiting factor [1]. Highly integrated systems-on-chip are common in this market and their memories typically occupy the largest fraction of the chip area, contributing to most of the overall cost. Even small increases in memory area translate directly to equivalent increases in cost, which lead to enormous levels of lost profit at large scales [2]. In such constrained scenarios, reducing the code size is essential [3].

A research group from the University of Edinburgh discovered Function Merging by Sequence Alignment (FMSA), a new way to merge two arbitrary functions. [4]. Their technique works on any two arbitrary functions.

B. Implementation of FMSA

The whole function merging task contains two main parts. The first part is choosing two possible profitable functions, which will be given in Section I-C in detail.

The second part is trying to merge two functions and check whether it is profitable to merge those two functions. The first step is linearization. FMSA makes the CFG of one certain function into a linear execution of instructions for the next merging step, as we can see in Fig. 1.

[1]Code available at https://github.com/lichye/Speedup-implements-FMSA

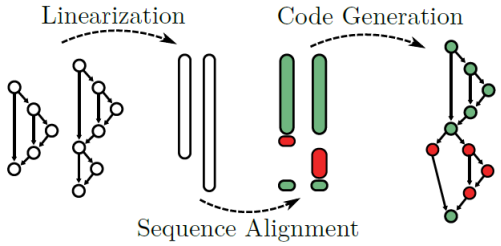

Fig. 1. Linearization

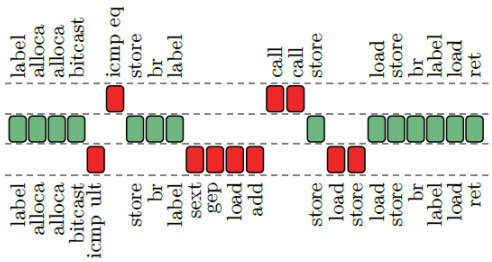

Fig. 2. Sequence Alignment

After linearization, FMSA reduces the problem of merging functions to the problem of sequence alignment. Sequence alignment is important in many scientific areas, most notably in molecular biology where it is used for identifying homologous subsequences of amino acids in proteins [5]. The main idea is to find the most similar parts of two sequences and pair those similar parts. After pairing, sequence alignment adds different parts into the output sequence. [4] This procedure can be viewed in Fig. 2.

C. Problem in FMSA

The first step of FMSA is to choose a possible function pair with profit. Although, we can merge any two arbitrary functions, we should choose profitable pairs to merge.

In the FMSA implementation, there are unavoidable calculations of Manhattan distance between each two functions. This can also be seen in Fig. 3. Hence, there will be Manhattan distance calculations in quadratic time. After calculating all the Manhattan distances, FMSA will pick up top-1, top-5, or top-10 pairs of functions with the lowest Manhattan distances to conduct merging attempts . In this way, FMSA implements

```
mergeFunctions(Funcs):
    Worklist = Funcs
    AvailableFuncs = Funcs

    while worklist is not empty:
        F1 = Worklist.pop()
        sort AvailableFuncs by Manhattan distance
        for i in 1...MaxExploration:
            F1 = AvailableFuncs[i]
            FMerged = mergeBySequenceAligment(F1,F2)
            if FMerged.size < F1.size + F2.size:
                update call graph
                delete F2 from AvailableFuncs
                add FMerged to Worklist
```

Fig. 3. Manhattan Distance

can be divided into fm1, fm5, and fm10. We will analyze the differences of three implementations in the later part.

In addition, the comparison is in the order of functions in the program,. Once there is a profitable merge, FMSA will merge instantly. This will influence precision of the grouping process as well.

Hence, we need to figure out a better way for grouping all the functions and selecting functions to be merged.

II. Proposed Method & Implementation

The primary function merging technique for this study is FMSA. We use the FMSA as the baseline, and implement a novel technique by modifying the process of finding prospective mergeable functions. More specifically, we replaced the pairwise ranking-based exploration mechanism of FMSA with a mechanism in a grouping manner.

In this section, We will discuss the feature (fingerprint) selection for similarity evaluation and 3 proposed methods.

A. Fingerprint Selection

As with the original method, our ranking mechanism also utilizes the fingerprint of the functions to vectorize functions and evaluate similarity. With the use of fingerprints, we are able to avoid high compilation time by filtering out the unpromising functions earlier. In contrast to the original method, we choose the frequency of instruction opcodes as our fingerprint. This is because only similar functions are worth merging, and they should have similar opcode counts. Since LLVM IR has about 64 opcodes, storing fingerprints as an integer vector of opcode frequency rather than tracking all instructions would help with the efficiency of similarity evaluation.

B. Proposed Methods

1) Linear Time Grouping with Reference Vector:

To find the best-case merge while avoiding quadratic time comparison, our first attempt is to propose the concept of *reference vector* and to compute the Manhattan distance with respect to a reference vector.

Manhattan distance can only be computed between two functions. However, we want to avoid pairwise comparison which leads to an increase in the compilation time. With the use of reference vector, we are able to do the linear time grouping to find mergeable functions. Reference vector ref is computed as the mean of all function $\{f_1, f_2, ..., f_n\}$ on the frequency of each opcode op:

$$ref(op) = (1/n)\sum_{i=1}^{n}(freq(op, f_i))$$

Then, we use this reference vector to compute the Manhattan distance of each function. At this point, each function has its own similarity score $mh(f_i)$. Suppose there are k opcodes.

$$mh(f_i) = \sum_{j=1}^{k} |freq(op_j, f_i) - freq(op_j, ref)|$$

To reduce the scope of comparison, we group the functions in linear time using a certain length of interval instead of doing pairwise comparisons. The length of the interval L is pre-defined. In our algorithm, we first sort functions by the similarity score. We then iterate through the sorted functions and group together all ungrouped functions with similarity scores within interval $[mh(f_{ungroup}), mh(f_{ungroup}) + L)$, where $f_{ungroup}$ is the first ungrouped function. Finally, we merge functions within each group.

Fig. 4 shows a simple example of 3 functions with only 4 opcodes. Each line is represented in the opcode frequencies. For example, the frequency of OP_0 of f_1. The last three lines represent a function for each. The first line in blue is the reference vector calculated by f_1, f_2, and f_3. With the use of reference vector, we have $mh(f_1) = 6$, $mh(f_2) = 7$, and $mh(f_3) = 11$. For future grouping, we set the length of interval as 1.5. Then, the functions in yellow are recognized as mergeable functions because they fall in $[6, 7.5)$.

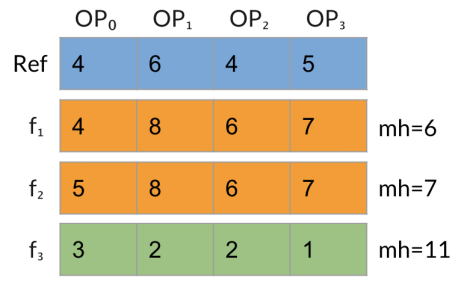

Fig. 4. Example of grouping functions using Manhattan Distance for merge

2) Cosine Similarity:

There is a potential problem in grouping with Manhattan distance. Manhattan distance does not capture the relationships between the features within a vector. Similarity in this case is simply computed using a crude summation of feature differences between a function vector and the reference vector. Thus, there are cases where it may fail at distinguishing functions that are not that similar as shown in Fig. 5. This

figure shows an example of n functions with only 4 opcodes. With an interval length of 1.5, f_1, f_2, and f_3 are grouped because their Manhattan distances are in $[8, 9.5)$. However, it is evidently shown that f_1 is not similar to f_2 and f_3. The pattern in the features of f_1 does not relate to those of the other two. Therefore, to address this issue, we need a new metric to reflect actual similarity better.

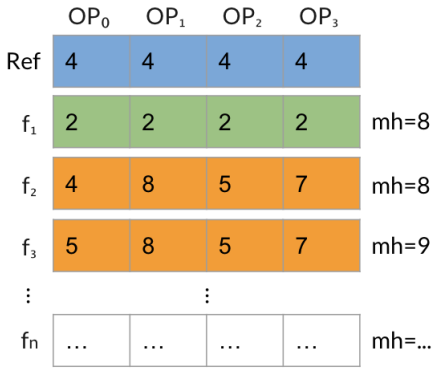

Fig. 5. Example of Manhattan Distance fails in grouping

Cosine similarity is a very common technique in the field of Natural Language Processing, whether it is for text classification or ranking documents for a search engine. It will first vectorize a document based on its word frequencies, and then compute similarity using the angle between the document vectors in the form of cosine value. For function merging, the context is similar, because function merging is also vectorizing the functions into fingerprint vectors based on the functions' opcode frequencies. Hence, cosine similarity can be easily applied to function merging. With cosine similarity, similarity is now computed based on the angle between the vectors, and the smaller the angle, the more similar they are.

Thus, we proposed a new method by replacing the Manhattan distance with cosine similarity in linear time grouping with reference vector. We implement the cosine similarity with the following equation:

$$cos(f_i) = \frac{\sum_{j=1}^{k} freq(op_j, f_i) \cdot freq(op_j, ref)}{\sqrt{\sum_{j=1}^{k} freq(op_j, f_i)^2} \sqrt{\sum_{j=1}^{k} freq(op_j, ref)^2}}$$

3) Code size choosing:

After an in-depth investigation on the functions our algorithm had merged, we observed that it made more unsuccessful attempts on small sized functions (with cost value less than 20) compared to the original method. Additionally, for some benchmarks such as 483.xalancbmk, our algorithm performed more merges compared to the original algorithm while failing to increase the actual code size reduction. Based on these two observations: 1) merging small-sized function is time-consuming and 2) in general not profitable, we propose to skip those small-sized functions to further reduce the compilation time spent on function merging. In our implementation we choose cost 14 as the threshold. It is chosen empirically so that most valid merges performed by our algorithm will not be excluded from consideration. Functions with cost less than 14 will not be added to the initial sorted list thus not being considered in subsequent function merging steps. With skipping small-sized functions, we are able to gain further speedup on compilation time with small loss on code size reduction.

III. Evaluation

A. Experiment setup

We compare our methods with the fm1 verison of FMSA. We chose fm1 as our baseline because it is the best-performing configuration in terms of compilation time. As shown in Fig. 6, fm1 is significantly faster than fm5 and fm10 up to 5.6x.

However, in order to guarantee that our solutions are up to par with FMSA, we also want to compare code size reduction in the interest of not sacrificing too much in that regard.

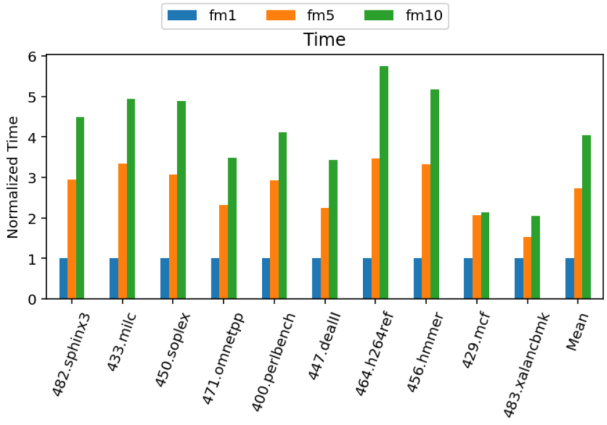

Fig. 6. Compilation time of all FMSA setups over all benchmarks

In the evaluation, we will discuss the results and performance of the three proposed methods: Manhattan distance (mh), cosine similarity (cosine), and cosine similarity without small functions (cosine-no-small-func). We collected the data based on three metrics: number of merges made, code size reduction, and compilation time. Percentage of code size reduction is calculated with respect to the unmerged code size. For this evaluation we will mainly focus on the latter two, but we will discuss some implications regarding the first at the end of this section.

We pre-defined the length of interval as 260 for mh and 0.05 for cosine. All implementations and optimizations are in LLVM v8 and are evaluated on the benchmark suite: C/C++ SPEC CPU2006 [6].

B. Manhattan Distance

We first evaluate the performance of mh, which compares functions with a reference fingerprint vector and uses Manhattan distance to the reference vector to group the functions. The results of mh compared to fm1 show that mh slightly underperforms fm1 in both code size reduction and compilation

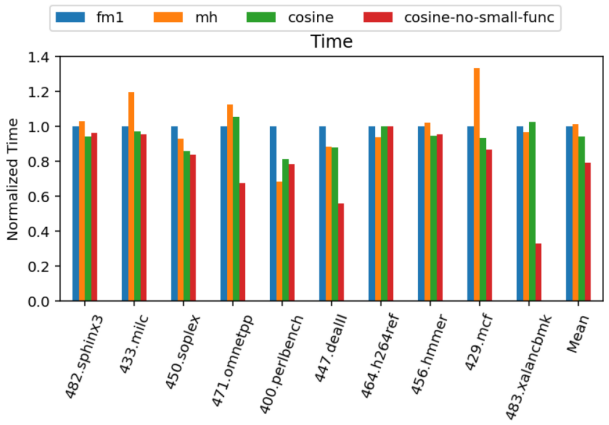

Fig. 7. Compilation time (normalized) across all benchmarks

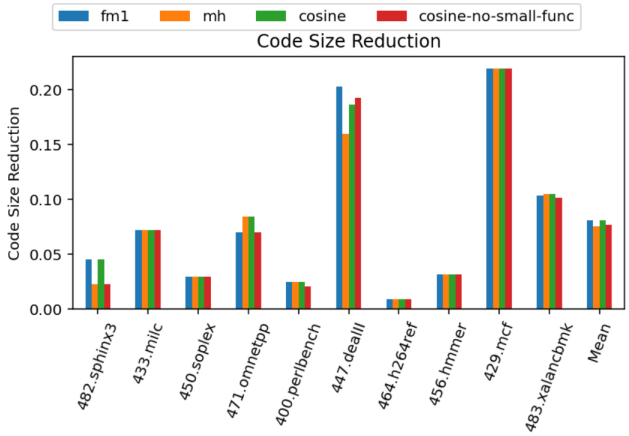

Fig. 8. Code size reduction % across all benchmarks.

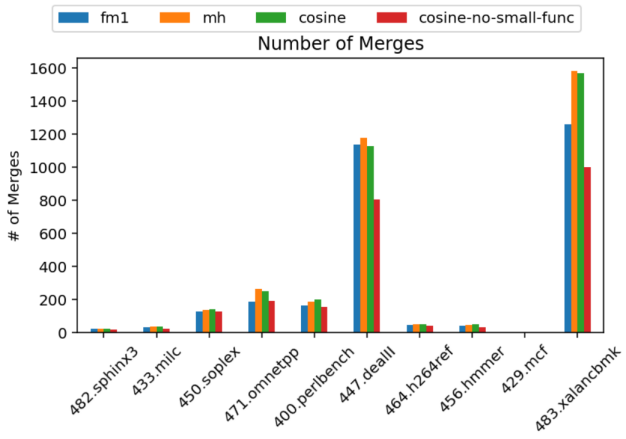

Fig. 9. Number of merges across all benchmarks

time. For most benchmarks, mh performed about the same as fm1, if not worse, for both code size reduction and compilation time. However, in particular with benchmark 447,dealII, mh underperforms fm1 greatly by about 4% in terms of code size reduction, and on average underperforms fm1 by about 0.5%. As for compilation time, mh took longer time than fm1 for benchmarks 433, 471, and 429. In particular with benchmark 429.mcf, mh's compilation time is about 0.33 ms (33%) longer than fm1. The average time for mh is a fringe 1 % longer than fm1's average compilation time.

C. Cosine Similarity

Next, we evaluate the performance of cosine similarity, which uses the cosine values between the vectors to assess function similarity. In terms of code size reduction, cosine similarity generally did as well as fm1. For benchmark 447.dealIII, cosine underperforms fm1 by about 1.4 % but the mean code size reduction across all benchmarks for cosine is almost identical to that of fm1 with a negligible difference. For compilation time, cosine similarity is able to overperform fm1 across most benchmarks, barring slight underperformance on 471.omnetpp and 483.xalancbmk. The mean normalized compilation time is shorter than that of fm1 by about 6 %.

D. Cosine Similarity no Small Functions

Now we take a look at the performance of cosine-no-small-func, which is cosine similarity but it ignores all small functions for comparisons. Cosine-no-small-func generally underperforms fm1 in terms of code size reduction, namely for benchmark 482.sphinix3 by 2.3%. The mean code reduction is about 0.4 % less that that of fm1. However, it gains the biggest win in compilation time, overperforming fm1 across all benchmarks and yielded a mean normalized compilation time of about 79 % of fm1's compilation time, faster than fm1's mean normalized compilation time by 21%, mh's mean normalized compilation time by 22%, and cosine's mean normalized compilation time by 15%.

E. Discussion

Overall, none of our methods technically improved code size reduction with respect to FMSA. This is a natural outcome because FMSA is willing to take the code size reduction vs. compilation time trade-off, and sacrifices compilation time performance so as to conduct a holistic search by doing pair-wise comparisons of the functions' fingerprint vectors.

Our methods, on the other hand, add another layer of abstraction, or heuristics, if you will, on top of these function vectors. We used their differences to a reference vector as well as a different interval to classify them into groups of "similar functions". The core idea is that we can apply this strategy to avoid excessive pair-wise comparisons of non-similar functions in the interest of saving time. Cosine-no-small-func takes this approach further by ignoring small functions altogether.

However, even when speaking of code size reduction, all proposed methods' mean values are within the residual range of 0.5% compared to the baseline mean value, so practically speaking the underperformance in code size reduction for these methods is not noteworthy. Nonetheless, we were able to achieve a win with cosine and cosine-no-small-func in

compilation time, of which cosine-no-small-func on average reduces compilation time by a relatively large margin.

Manhattan distance is the worst performing method of the three. This is likely due to an intrinsic flaw of using Manhattan distance as a metric to measure similarity with respect to a certain reference vector. This flaw could potentially hinder it from discerning dissimilar functions from similar ones, as shown in Fig. 5.

An important observation we made from the data collected is the seemingly non-existent correlation between merge times and code size reduction, as shown in Fig. 8 and Fig. 9. This led to the notion that not all functions are worth merging, and in particular small functions most likely would not impact code size reduction that much. Therefore, we applied a heuristics that ignores small function for cosine, and cosine-no-small-func ends up with the best compilation time performance for a very small price in code size reduction.

IV. Conclusion

A. Summary of Our Work

In this work, we proposed three methods for speeding up FMSA: Manhattan Distance, Cosine, and Cosine No Small Functions. Out of the three, Cosine No Small Functions is able to achieve a significant reduction in compilation time, with only a minuscule decrease in code size reduction compared to the baseline. It is able to achieve such performance by:

- Comparing functions to a single reference vector instead of conducting pair-wise comparisons.
- Ignoring small functions that are not worth merging.

B. Future Potential Work

In the future, the potential work can be considered in these directions.

- Do sequence alignment on more than two functions as we have found a large group with more than two functions.
- Implement a better Linearization Algorithm as the now-time algorithm just choose one possible Linearization.

Acknowledgements

This document is derived from previous conferences, in particular ISCA 2019, MICRO 2019, ISCA 2020, MICRO 2020, ISCA 2021, HPCA 2022 and ISCA 2022. We thank the organizers of MICRO 2022 for the idea to set up HotCRP topic selections.

References

[1] P. Plaza, E. Sancristobal, G. Carro, M. Castro, and E. Ruiz, "Wireless development boards to connect the world," in *Online Engineering & Internet of Things*, pp. 19–27, Springer, 2018.

[2] T. J. Edler von Koch, I. Böhm, and B. Franke, "Integrated instruction selection and register allocation for compact code generation exploiting freeform mixing of 16-and 32-bit instructions," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 180–189, 2010.

[3] U. P. Schultz, K. Burgaard, F. G. Christensen, and J. L. Knudsen, "Compiling java for low-end embedded systems," *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 42–50, 2003.

[4] R. C. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 149–163, IEEE, 2019.

[5] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[6] SPEC, "Standard performance evaluation corp benchmarks," 2014. http://www.spec.org.