CGO: G: Multiple Function Merging for Code Size Reduction

Yuta Saito, Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa
Waseda University
Tokyo, Japan
ysaito@suou.waseda.jp,exkazuu@gmail.com,washizaki@waseda.jp,fukazawa@waseda.jp

ABSTRACT

Program code size is important for resource-constrained environments such as embedded devices. An optimization technique known as function merging, which merges similar functions into a single function, has been developed. However, in the state-of-the-art approach, the number of functions that can be merged at once is limited to two, thus it is impossible to efficiently merge three or more functions, which are often generated from polymorphic functions. In this study, we propose multiple function merging as an approach for merging three or more similar functions into a single function; we also evaluate it using SPEC CPU2006. The results show that it reduces code size by as much as 3.78% compared with the reduction achieved with the state-of-the-art approach.

1 INTRODUCTION

Code size is critical for resource-constrained environments such as Internet of Things (IoT) devices and programs delivered over networks [1, 2]. Practical compilers apply some optimizations to reduce code size, and merging functions is one of the promising approaches to reducing code redundancy.

According to a study of the state-of-the-art method of fast focused function merging (F3M) [3], function merging consists of a matching phase to find similar functions, an alignment phase, and a code generation phase. In existing methods, two or more similar functions are found in the matching phase, whereas the input for the alignment phase and later phases is limited to two functions. However, practical programming languages such as C++ have language features for polymorphic functions and perform monomorphization, which can replicate a polymorphic function resulting in the synthesis of many similar functions [4]. For example, Figure 1 shows that the existing 2-merge methods fail to merge three similar functions into a single function, whereas our proposed multiple-merge method successfully merges them. The existing methods initially select and merge two of the three functions; however, the merged function and the remaining function are dissimilar, preventing the merge. By contrast, our proposed method can merge the three functions directly.

We hypothesize that the existing two-function limitation misses many optimization opportunities and that unlocking the limitation could achieve a further reduction in code size. Under this hypothesis, we extended the implementation of the alignment and code generation phases to accommodate more than three functions based on LLVM [5]. We refer to this idea as multiple function merging.

The research questions addressed in this study are as follows:

- RQ1: How much code size reduction can be achieved by the proposed approach compared with that achieved by F3M?
- RQ2: How many more merges can be achieved by the proposed approach compared with the F3M?

The evaluation results show a reduction as large as 3.78% in SPEC2006 compared with the reduction achieved with the state-of-the-art approach. In addition, the evaluation showed that the proposed approach can merge 1.51 times more functions. With this study, we make the following contributions:

- The first approach to merge an arbitrary number of SSA-form functions at once is proposed. The implementation is publicly available as an LLVM plugin on GitHub [1].
- The effectiveness of multiple function merging for practical applications is demonstrated.

2 BACKGROUND

This section explains the two existing function merging optimizations.

Identical function merging is a relatively simpler optimization that merges only identical functions into a single function. It is a well-known optimization method implemented in many practical compilers in various ways and in various build phases such as compilers or linkers.

Function merging by sequence alignment is a more complicated and general optimization that merges similar functions into a single function. Function merging by sequence alignment comprises roughly three steps: function pairing, sequence alignment, and code generation. The first step, function pairing, is conducted to find similar functions that might be merged. The second step, sequence alignment, then attempts to find the common sequence of instructions among the similar functions identified in the first step. Finally, the third step, code generation, generates the merged function from the aligned instruction sequence while preserving the input function semantics.

Compilers often artificially generate such nearly identical functions from polymorphic functions. A parametric polymorphic function is a function that can evaluate or be applied to values of different types [6]. Although polymorphic functions can be implemented using several different approaches, practical compilers for C++ and Rust often clone the function body for each type and generate a function for the specific type for runtime performance. This cloning is called *instantiation* or *monomorphization* [4].

The number of clone instances can be more than two for such languages. Therefore, methods to efficiently merge three or more functions would be useful.

3 RELATED WORK

Although initially motivated by performance, many intra-procedural optimization techniques improve performance by reducing code size. Having a smaller number of executed instructions can reduce the number of cache misses and improve the performance of the

[1] https://github.com/kateinoigakukun/llvm-next-function-merging

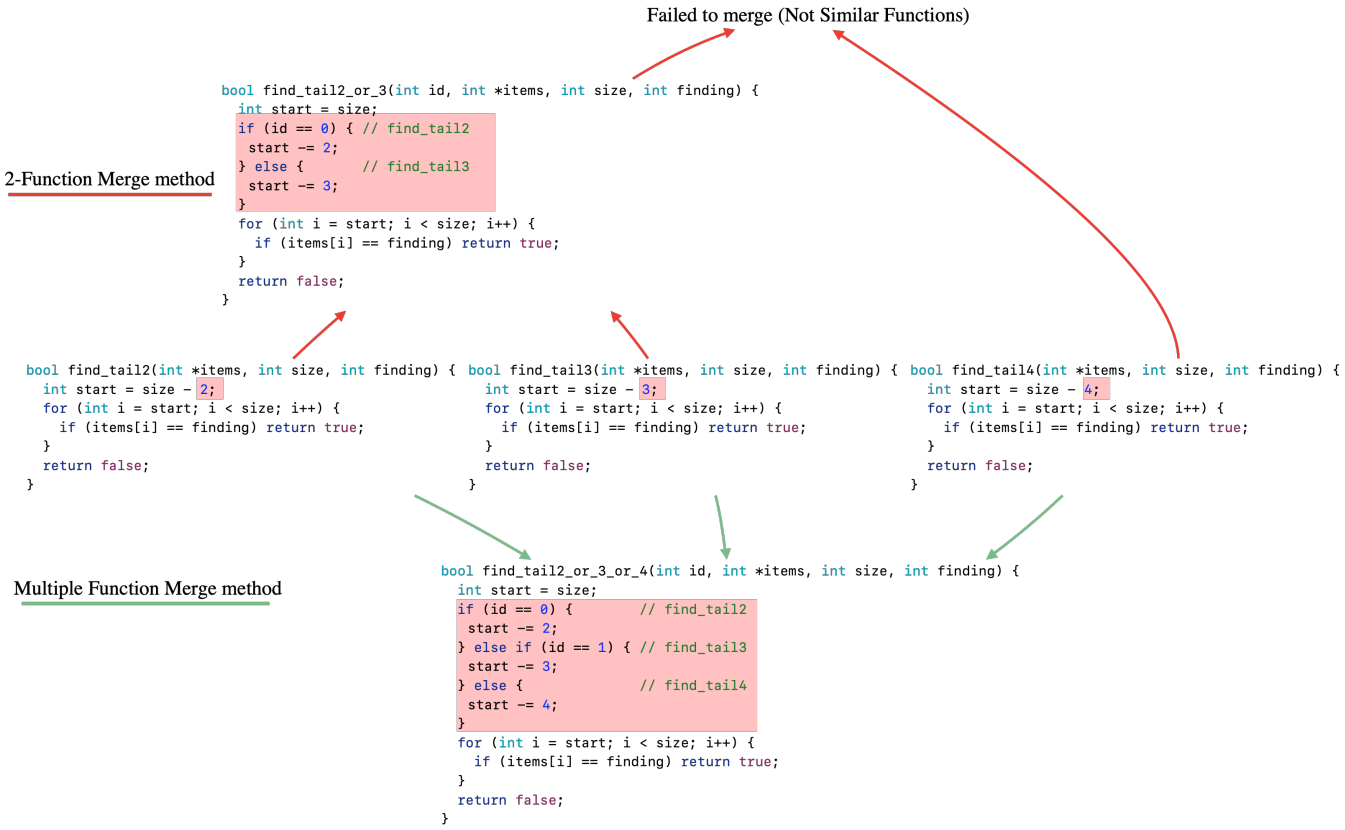

Figure 1: Merging similar functions by the existing two-function merging method and the multiple function merging method

processor. Optimizations that effectively reduce code size include certain kinds of strength reduction and the removal of redundant code and dead code, and certain kinds of strength reduction [7, 8].

Intra-procedural optimizations are effective in reducing code size; however they have limited opportunities to find redundancies in programs. Interprocedural optimizations can find more redundancies in programs by analyzing the whole program.

Dead function elimination is a compiler optimization that literally removes dead functions from the program [9]. Outlining reduces code size by moving a part of common code into a separate function while sacrificing performance [10, 11].

Function merging focuses on function-level merging, outlining focuses on basic block-level merging, and other intra-procedural optimizations focus on instruction-level merging. Therefore, all these strategies are orthogonal and could be used in a complementarily.

4 APPROACH AND UNIQUENESS

4.1 Overview

Function merging consists of roughly three steps: function paring, sequence alignment, and code generation. The state-of-the-art function merging technique, F3M [3], already finds a set of similar two or more functions. However, it selects only two functions from the set and merges them. The idea proposed here is to select more than two functions from the set, align their instruction sequences, and merge them into a single function at once.

4.2 Multiple Instruction Sequence Alignment

The goal of the multiple instruction sequence alignment is to identify the common parts of the instruction sequences of the functions in the set found in the first paring step. That is, the goal is to find the different parts that must be selectively executed based on which of the input functions has been called in the original code. The prior arts (e.g. F3M) reduces the problem to the sequence alignment problem by linearizing the control-flow graphs (CFGs) to instruction sequences. The linearization process takes a CFG of a function, traverses the CFG in breadth-first order, and generates an instruction sequence from the traversed CFG nodes. After the linearization, the problem can be reduced to the sequence alignment problem, which is a well-known problem in the field of bioinformatics.

The proposed approach extends the prior arts' idea of linearizing the CFGs and using the sequence alignment algorithm by considering multiple instruction sequences. The multiple sequence alignment [12] algorithm used in the proposed approach is a variant of the Needleman-Wunsch algorithm [13], which is a dynamic programming algorithm for sequence alignment. The proposed algorithm extends the Needleman-Wunsch algorithm to take multiple sequences as the input and computes scores for each possible

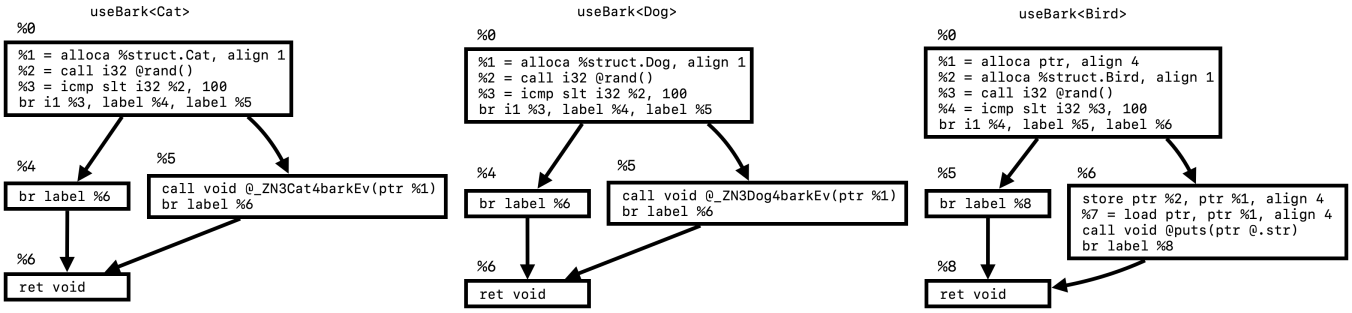

Figure 2: Example set of the input function CFGs

Mergeable Non-Mergeable

useBark<Cat>	useBark<Dog>	useBark<Bird>
Basic Block %0	***Basic Block %0***	***Basic Block %0***
		%1 = alloca ptr
		%2 = alloca %struct.Bird
	%1 = alloca %struct.Dog	
%1 = alloca %struct.Cat		
%2 = call i32 @rand()	%2 = call i32 @rand()	%3 = call i32 @rand()
%3 = icmp slt i32 %2, 100	%3 = icmp slt i32 %2, 100	%4 = icmp slt i32 %3, 100
br i1 %3, label %4, label %5	br i1 %3, label %4, label %5	br i1 %4, label %5, label %6
Basic Block %4	***Basic Block %4***	***Basic Block %5***
br label %6	br label %6	br label %8
Basic Block %5	***Basic Block %5***	***Basic Block %6***
		store ptr %2, ptr %1
		%7 = load ptr, ptr %1
		call @puts(ptr @.str)
	call @_ZN3Dog4barkEv(ptr %1)	
call @_ZN3Cat4barkEv(ptr %1)		
br label %6	br label %6	br label %8
Basic Block %6	***Basic Block %6***	***Basic Block %8***
ret void	ret void	ret void

Figure 3: Example of an aligned three-function instruction sequence. Each column is an instruction sequence of a function with gaps.

alignment of the sequences in N-dimensional space, where N is the number of input sequences.

Figure 3 shows the result of the alignment against the example set of functions shown in Figure 2. Each column in the figure shows an instruction sequence of a function with gaps. A green row indicates that instructions in the row are mergeable into a single function, whereas a red cell indicates that it is non-mergeable with other instructions. A white cell represents a gap inserted by the alignment.

Better approaches to the Needleman-Wunsch algorithm in terms of computational complexity and memory usage have been developed[14]. However, the Needleman-Wunsch approach guarantees finding the globally optimal alignment, and the quality of the alignment is more important than the efficiency when evaluating the potential reduction in code size.

4.3 Code Generation

The multiple instruction sequence alignment provides a set of aligned instruction sequences. The next step is to generate a new function from the aligned sequence. The underlying idea of code generation is the same as that in the prior art F3M. The main difference between our method and the prior art is that our code generation is more general with respect to the number of input functions while maintaining the same semantics of the original functions. More specifically, our code generation can handle differences in

- Function parameters
- CFG layout
- Instructions in its basic blocks

4.3.1 Layout of Merged Function Parameters. To accommodate the difference in function parameters, we employ the same approach used in the prior arts except that we change the bit width of the function identifier according to the number of input functions. The function identifier is a parameter of the merged function, and it is used to select the correct path to execute on the basis of which function the merged function pretends to be. The bit width of the function identifier is always 1 in the prior arts because the number of input functions is always 2. However, our code generation can accommodate more than two functions, and the number is not fixed. Therefore, we determine the bit width of the function identifier to be $\log_2$ of the number of input functions.

After the function identifier, the parameters of the original functions are placed in the merged parameter list in the same manner as in the prior arts. At the same time, it attempts to minimize the number of parameters by reusing parameters that have the same type and disjoint definitions among input functions.

4.3.2 Layout of Merged Function CFG. After the parameter merging, the next step is to layout the CFG of the merged function. The basic layout algorithm of the CFG is the same as the one used in the prior art F3M, and we extended it to accommodate more than two input functions. Figure 4 shows the CFG of the merged function produced by the code generation algorithm against the example set of functions shown in Figure 2.

4.3.3 Operand Assignment. After all instructions and basic blocks have been generated, the next step is to assign the operands of the instructions. The operand assignment has two steps. First, it assigns all label operands of the generated instructions. Second, it assigns the remaining operands of the instructions. The label operands are assigned first to construct the remaining CFG edges of the merged function. After the label operands are assigned, the complete CFG of the merged function is attained; the code generator can then create a dominator tree of the merged function. The dominator tree is used to determine where the non-label operands are selected while maintaining the dominance property of the SSA form.

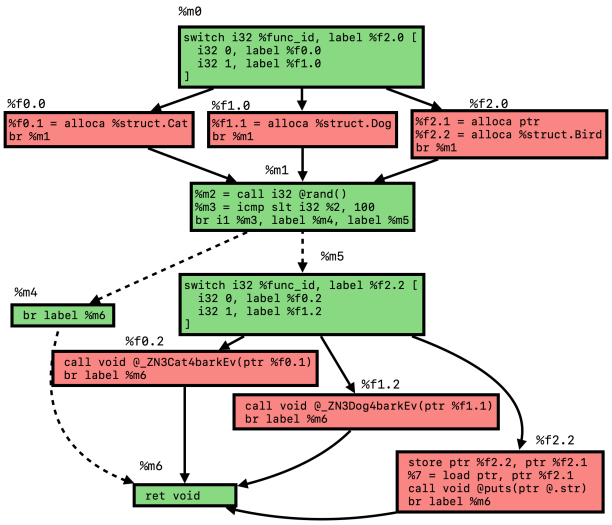

Figure 4: Merged CFG produced by the code generation step. Green blocks are shared basic blocks among input functions, and red blocks are non-mergeable basic blocks.

To assign the label operands, the code generator iterates over the aligned instruction sequence, and for each alignment entry, the code generator emits extra instructions to select the correct operand don't he basis of the function identifier if the corresponding operands are merged.

In LLVM, labels are used to express the control flow of the program. The label operands of the terminator instructions such as `br` and `switch` are used to express the edges of the CFG. If the merged terminator instruction in the aligned instruction sequence has a label operand and the operands of the corresponding instructions in the input functions are different, the code generator emits a basic block that has a `switch` instruction and it jumps into the correct basic block based on the function identifier. If the operands are the same among the aligned instructions, the code generator simply inserts the original operand into the merged instruction.

As an example, Figure 5 shows the code generated to handle different label operands of the `br` instruction in the aligned instruction sequence. In the figure, the `br` instruction is merged from three `br` instructions having destination labels $\%f0.1$, $\%f1.4$, and $\%f2.8$. The code generator generates a new basic block named *%selector* and it puts a `switch` instruction jumping into $\%f0.1$, $\%f1.4$, and $\%f2.8$ based on the given function identifier. The code generator then assigns the *%selector* label as the operand of the `br` instruction.

To assign the remaining non-label operands, the code generator iterates over the aligned instruction sequence, and for each alignment entry, the code generator emits extra instructions to select the correct operand based on the function identifier as well as the label operand assignment. The code generator emits a `switch` instruction, empty basic blocks for each input function, and an aggregation basic block that has a `phi` instruction to select the correct operand based on the incoming basic block, which is determined by the function identifier `switch` instruction used. Figure 6 shows the code generated to handle different non-label operands of the `add` instruction in the aligned instruction sequence.

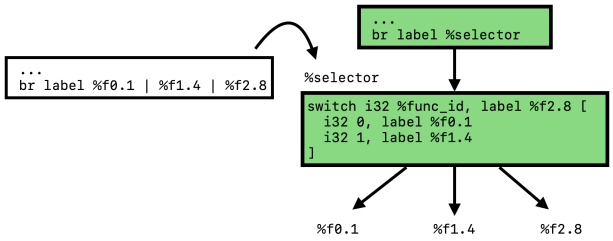

Figure 5: Example of label selection for the `br` instruction

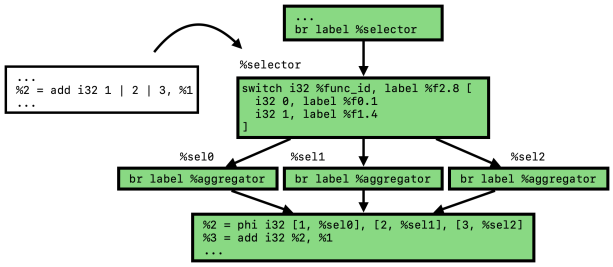

Figure 6: Example of value operand selection for the `add` instruction

5 RESULTS AND CONTRIBUTIONS

We ran the all experiments on a dedicated server with an 8 core AMD Ryzen 9 5900HX CPU, 64 GB of RAM, and Ubuntu 20.04.4 LTS. We used the same LLVM LTO compilation pipeline as the prior work [3]. The pipeline accepts a whole program as an LLVM IR module by linking all input bitcode files and then performs each function merging optimization pass. The LLVM version was 13.0.0.

We evaluated both the proposed approach and the F3M approach on SPEC CPU2006. The proposed approach was implemented as an LLVM pass plugin, with a configurable parameter e, which is the maximum number of functions that can be merged into a single function at once.

Table 2: Average metrics for each technique

Technique	Reduction rate [a]	# of MF [b]	CT overhead [c]
F3M	19.90%	12.30%	32.12%
MFM [e=3]	20.45%	14.86%	1796.35%
MFM [e=4]	20.26%	15.84%	15620.12%

[a] Reduction rate compared with the size of the whole program without any function merging optimization.
[b] Number of merged functions
[c] Compile time overhead

5.1 Related to the Research Questions

We measured these three metrics for F3M and our proposed approach for $e = 3$ and 4: the object file size, the number of merged functions, and the compile time. Table 1 shows the all results for each metric.

RQ1: How much code size reduction can be achieved by the proposed approach compared with that achieved by F3M?. Table 2 shows that the proposed approach outperforms F3M on average in terms of code size reduction. In addition, the result of the `471.omnetpp` shows that the proposed approach outperforms F3M by as much as 3.78% (reducing code size from 714.38 kB to 687.36 kB) when

Table 1: Average metrics for each technique

	Object size reduction rate			% of merged function			Compile time increase rate		
Benchmark	F3M	MFM[a] (e=3)	MFM (e=4)	F3M	MFM (e=3)	MFM (e=4)	F3M	MFM (e=3)	MFM (e=4)
483.xalancbmk	27.682%	28.090%	28.100%	7.558%	9.672%	10.301%	114.510%	1444.372%	14606.497%
471.omnetpp	11.203%	14.561%	13.369%	23.549%	32.827%	33.044%	19.660%	1152.366%	20690.159%
462.libquantum	23.110%	23.373%	21.379%	25.333%	24.667%	24.000%	104.078%	757.728%	10757.235%
456.hmmer	9.460%	9.640%	9.744%	26.059%	27.850%	28.502%	130.793%	2305.937%	21042.939%
445.gobmk	2.721%	2.990%	2.476%	28.488%	33.163%	37.984%	137.714%	2170.487%	18123.516%
444.namd	20.291%	20.881%	20.482%	33.333%	34.483%	36.782%	179.478%	754.392%	1569.829%
401.bzip2	7.029%	6.671%	7.163%	30.400%	30.400%	30.400%	196.840%	1735.483%	10717.908%
400.perlbench	7.593%	8.371%	7.915%	40.061%	40.819%	42.438%	164.902%	3543.803%	19819.889%

[a] Multiple function merging

$e = 3$. In summary, these results show that the proposed approach can achieve a non-trivial reduction in code size for real-world programs. However, a 4-merge ($e = 4$) adversely affected on the size on average because of the poor accuracy of similar function paring and instruction sequence alignment.

RQ2: How many more merges can be achieved by the proposed approach compared with the number achieved by F3M?. Table 2 shows that, on average, the proposed approach outperforms F3M in terms of the number of merged functions. For the `471.omnetpp`, F3M merged 434 functions, whereas the proposed approach merged 609 functions when $e = 4$ and the total number of functions in the input program was 1843. These results indicate that the proposed approach merged, at most, 1.40 times more functions than F3M. Therefore, the proposed approach achieves a non-trivial number of more merges than F3M.

5.2 Limitations and Threats to External Validity

The proposed approach cannot align large-scale instruction sequences because of poor computational complexity and memory space complexity. Also, the benchmark suites we used are smaller than real-world programs because the proposed approach cannot yet align large-scale instruction sequences. Therefore, the results of the evaluation might not be representative of real-world programs.

6 CONCLUSION AND FUTURE WORK

We have presented a compiler-based function merging approach, multiple function merging, to reduce code size by unlocking three or more functions to be merged into a single function. Unlike the existing state-of-the-art approach F3M, the proposed approach does not limit the number of functions that can be merged by its code generation algorithm. As a result, the proposed approach can merge functions efficiently. We implemented the proposed approach as an LLVM pass plugin and evaluated it on several benchmark suites including real-world programs. The experimental results show that the proposed approach outperforms the previous state-of-the-art approach in terms of code size reduction, which is 3.78% at most.

In future work, we plan to investigate the trade-off relationship between the code size reduction rate, the compilation time, and memory usage to make the compilation time acceptable by using a non-globally optimal instruction sequence alignment algorithm used in the bioinformatics field or the basic block level instruction alignment proposed as HyFM [15].

ACKNOWLEDGEMENTS

We thank Prof. Washizaki, Prof. Fukazawa, and Assoc. Prof. Sakamoto for their supervision and assistance.

REFERENCES

[1] Y. Sakamoto, S. Matsumoto, S. Tokunaga, S. Saiki, and M. Nakamura, "Empirical study on effects of script minification and http compression for traffic reduction," in *2015 Third International Conference on Digital Information, Networking, and Wireless Communications (DINWC)*, 2015, pp. 127–132.

[2] R. Auler, C. E. Millani, A. Brisighello, A. Linhares, and E. Borin, "Handling iot platform heterogeneity with coisa, a compact openisa virtual platform," *Concurrency and computation*, vol. 29, no. 22, pp. e3932–n/a, Nov 25, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3932

[3] S. Stirling, R. R. C. O., K. Hazelwood, H. Leather, M. O'Boyle, and P. Petoumenos, "F3m: Fast focused function merging," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 242–253.

[4] D. A. Wood, "Polymorphisation: Improving rust compilation times through intelligent monomorphisation," 2020.

[5] C. Lattner and V. Adve, "Llvm," ser. CGO '04. IEEE Computer Society, Mar 20, 2004, pp. 75–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=977673

[6] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM computing surveys*, vol. 17, no. 4, pp. 471–523, Dec 10, 1985. [Online]. Available: http://dl.acm.org/citation.cfm?id=6042

[7] P. BRIGGS, K. D. COOPER, and L. T. SIMPSON, "Value numbering," *Software, practice and experience*, vol. 27, no. 6, pp. 701–724, Jun 1997. [Online]. Available: https://api.istex.fr/ark:/67375/WNG-67H10B30-Q/fulltext.pdf

[8] S. Debray, W. Evans, R. Muth, and B. D. Sutter, "Compiler techniques for code compaction," *ACM transactions on programming languages and systems*, vol. 22, no. 2, pp. 378–415, Mar 1, 2000. [Online]. Available: http://dl.acm.org.waseda.idm.oclc.org/citation.cfm?id=349233

[9] T. J. W. I. R. C. R. Division, F. Allen, and J. Cocke, "A catalogue of optimizing transformations." [Online]. Available: https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf

[10] P. Zhao and J. N. Amaral, "Function outlining and partial inlining," vol. 37. Chichester, UK: John Wiley and Sons, Ltd, Apr 25, 2007, pp. 465–491. [Online]. Available: https://api.istex.fr/ark:/67375/WNG-F8CPT804-7/fulltext.pdf

[11] M. Chabbi, J. Lin, and R. Barik, "An experience with code-size optimization for production ios mobile applications," in - *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 363–377, iD: 1.

[12] H. CARRILLO and D. LIPMAN, "The multiple sequence alignment problem in biology," *SIAM journal on applied mathematics*, vol. 48, no. 5, pp. 1073–1082, Oct 1, 1988. [Online]. Available: https://www.jstor.org/stable/2101469

[13] 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. [Online]. Available: https://dx.doi.org/10.1016/0022-2836(70)90057-4

[14] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *Journal of computational biology*, vol. 1, no. 4, pp. 337–348, 1994. [Online]. Available: https://www.liebertpub.com/doi/abs/10.1089/cmb.1994.1.337

[15] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, K. Hazelwood, and H. Leather, "Hyfm: Function merging for free," Apr 09, 2021. [Online]. Available: https://www.research.manchester.ac.uk/portal/en/publications/hyfm-function-merging-for-free(c8af3c99-0892-472d-aa5d-01f8a1d422ff).html