

Function Merging by Sequence Alignment

Rodrigo C. O. Rocha
University of Edinburgh, UK
r.rocha@ed.ac.uk

Pavlos Petoumenos
University of Edinburgh, UK
ppetoume@inf.ed.ac.uk

Zheng Wang
Lancaster University, UK
z.wang@lancaster.ac.uk

Murray Cole
University of Edinburgh, UK
mic@inf.ed.ac.uk

Hugh Leather
University of Edinburgh, UK
hleather@inf.ed.ac.uk

Abstract—Resource-constrained devices for embedded systems are becoming increasingly important. In such systems, memory is highly restrictive, making code size in most cases even more important than performance. Compared to more traditional platforms, memory is a larger part of the cost and code occupies much of it. Despite that, compilers make little effort to reduce code size. One key technique attempts to merge the bodies of similar functions. However, production compilers only apply this optimization to identical functions, while research compilers improve on that by merging the few functions with identical control-flow graphs and signatures. Overall, existing solutions are insufficient and we end up having to either increase cost by adding more memory or remove functionality from programs.

We introduce a novel technique that can merge arbitrary functions through sequence alignment, a bioinformatics algorithm for identifying regions of similarity between sequences. We combine this technique with an intelligent exploration mechanism to direct the search towards the most promising function pairs. Our approach is more than 2.4x better than the state-of-the-art, reducing code size by up to 25%, with an overall average of 6%, while introducing an average compilation-time overhead of only 15%. When aided by profiling information, this optimization can be deployed without any significant impact on the performance of the generated code.

Index Terms—Code Size, Function Merging, IPO, LTO.

I. INTRODUCTION

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]. Just adding more memory is not always a viable option. 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], [4], [5], [6], [7]. Unfortunately, production compilers offer little help beyond dead-code elimination or merging identical functions [8], [9], [10]. Developers might have more luck just removing functionality from their libraries [6] or hand-optimizing their code [11].

Function merging reduces replicated code by combining multiple identical functions into a single one [12], [10]. Although a simple and intuitive concept, it is crucial for making high-level abstractions usable, when they introduce duplicate code [8], [9]. For example, some C++ ABIs may end up creating multiple identical constructors and destructors of a class to use in different contexts [9] and C++ templates replicate code for different specializations [8], [10]. More advanced approaches [13] have extended this idea into merging non-identical functions by leveraging structural similarity. Functions with identical control-flow graphs (CFGs) and only small differences within corresponding basic blocks are merged into a single function that maintains the semantics of the original functions. This is particularly important for handling specialized template functions with small differences in their compiled form.

While an improvement, even the state-of-the-art often usually fails to produce any noticeable code size reduction. In this paper, we introduce a novel way to merge functions that overcomes the major limitations of the state-of-the-art. Our insight is that the weak results of existing function merging implementations are not due to the lack of duplicate code but due to the rigid, overly restrictive algorithms they use to find duplicates.

Our approach¹ is based upon the concept of sequence alignment, developed in bioinformatics for identifying functional or evolutionary relationships between different DNA or RNA sequences. Similarly, we use sequence alignment to find areas of functional similarity in arbitrary function pairs. Aligned segments with equivalent code are merged. The remaining segments where the two functions differ are added to the new function too but have their code guarded by a function identifier. This approach leads to significant code size reduction.

Applying sequence alignment to all pairs of functions is prohibitively expensive even for medium sized programs. To counter this, our technique is integrated with a ranking-based exploration mechanism that efficiently focuses the search to

¹Artifact available at: <https://doi.org/10.6084/m9.figshare.7473260>

```

glist_t glist_add_float32(glist_t g, float32 val){
  gnode_t *gn;
  gn = (gnode_t *) mymalloc (sizeof(gnode_t));
  gn->data.float32 = val;
  gn->next = g;
  return ((glist_t) gn);
}

glist_t glist_add_float64(glist_t g, float64 val){
  gnode_t *gn;
  gn = (gnode_t *) mymalloc (sizeof(gnode_t));
  gn->data.float64 = val;
  gn->next = g;
  return ((glist_t) gn);
}

-----Merged Function-----
glist_t merged(bool func_id,
               glist_t g, float32 v32, float64 v64){
  gnode_t *gn;
  gn = (gnode_t *) mymalloc (sizeof(gnode_t));
  if (func_id)
    gn->data.float32 = v32;
  else
    gn->data.float64 = v64;
  gn->next = g;
  return ((glist_t) gn);
}

```

Fig. 1. Example of two functions from the benchmark `sphinx` with different parameters that could be merged, as shown at the bottom. We highlight where they differ.

the most promising pairs of functions. As a result, we achieve our code size savings while introducing little compilation-time overhead.

Compared to identical function merging, we introduce extra code to be executed, namely the code that chooses between dissimilar sequences in merged functions. A naive implementation could easily hurt performance, e.g by merging two hot functions with only few similarities. Our implementation can avoid this by incorporating profiling information to identify blocks of hot code and effectively minimize the overhead in this portion of the code.

In this paper, we make the following contributions:

- We are the first to allow merging arbitrary functions, even ones with different signatures and CFGs.
- A novel ranking mechanism for focusing inter-procedural optimizations to the most profitable function pairs.
- Our function merging by sequence alignment technique is able to reduce code size by up to 25% on Intel and 30% on ARM, significantly outperforming the state-of-the-art, while introducing minimal compile-time and negligible run-time overheads.

II. MOTIVATION

In this section we make the argument for a more powerful function merging approach. Consider the examples from two SPEC CPU2006 benchmarks shown in Figures 1 and 2.

Figure 1 shows two functions from the `482.sphinx3` benchmark. The two functions are almost identical, only their function arguments are of different types, `float32` and `float64`, causing a single operation to be different. As shown at the

bottom of Figure 1, these functions can be easily merged in three steps. First, we expand the function argument list to include the two parameters of different types. Then, we add a function identifier, `func_id`, to indicate which of the two functions is called. Finally, we place the lines that are unique to one of the functions in a conditional branch predicated by the `func_id`. Overall, merging these two functions reduces the total number of machine instructions by 18% in the final object file for the Intel x86 architecture.

Despite being so similar, neither GCC nor LLVM can merge the two functions. They can only handle identical functions, allowing only for type mismatches that can be removed by lossless bitcasting of the conflicting values. Similarly, the state-of-the-art [13], while more powerful, cannot merge the two functions either. It requires both functions to have the same list of parameters.

Figure 2 shows another two functions extracted from `462.libquantum`. While these two functions have the same signature, i.e. the same return type and list of parameters, they differ slightly in their bodies. Merging them manually is straightforward, shown at the bottom of Figure 2, reducing the number of instructions by 23% in the final object file. But again, none of the existing techniques can merge the two functions. The state-of-the-art can work with non-identical functions, but it needs their CFGs to be identical. Even a single extra basic block, as in this case, makes merging impossible.

These examples show that all existing techniques are severely limited. Optimization passes in production compilers work only on effectively identical functions. State-of-the-art techniques can merge functions only when they are structurally identical, with isomorphic CFGs, and identical signatures. All of them miss massive opportunities for code size reduction. In the next sections, we show a better approach which removes such constraints and is able to merge arbitrary functions, when it is profitable to do so.

III. OUR APPROACH

In this section we describe our proposed function merging technique and show how it merges the motivating examples. Our technique works on any two arbitrary functions, even when they have few similarities and merging them would be counter-productive. For that reason, we also introduce a cost model to decide when it is beneficial to merge two functions (see Section IV-A). To avoid an expensive quadratic exploration, we integrate our profitability analysis with an efficient ranking mechanism based on a lightweight fingerprint of the functions.

A. Overview

Intuitively, when we are manually merging two functions, in a textual format, we try to visualize them side by side, identifying the equivalent segments of code and the non-equivalent ones. Then, we use this understanding to create the merged function. In this paper, we propose a technique that follows this simple yet effective principle. At the core of our technique lies a sequence alignment algorithm, which is

```

void quantum_cond_phase(
int control, int target, quantum_reg *reg){
    int i;
    COMPLEX_FLOAT z;
    if(quantum_objcode_put(COND_PHASE, control, target))
        return;
    z = quantum_cexp(pi / (1 << (control - target)));
    for(i=0; i<reg->size; i++) {
        if(reg->node[i].state & (1 << control)) {
            if(reg->node[i].state & (1 << target))
                reg->node[i].amplitude *= z;
        }
    }
    quantum_decohere(reg);
}

void quantum_cond_phase_inv(
int control, int target, quantum_reg *reg){
    int i;
    COMPLEX_FLOAT z;

    z = quantum_cexp(-pi / (1 << (control - target)));
    for(i=0; i<reg->size; i++) {
        if(reg->node[i].state & (1 << control)) {
            if(reg->node[i].state & (1 << target))
                reg->node[i].amplitude *= z;
        }
    }
    quantum_decohere(reg);
}
-----Merged Function-----
void merged(bool func_id,
int control, int target, quantum_reg *reg){
    int i;
    COMPLEX_FLOAT z;
    if(func_id)
        if(quantum_objcode_put(COND_PHASE, control, target))
            return;
    float var = (func_id)?pi:(-pi);
    z = quantum_cexp(var / (1 << (control - target)));
    for(i=0; i<reg->size; i++) {
        if(reg->node[i].state & (1 << control)) {
            if(reg->node[i].state & (1 << target))
                reg->node[i].amplitude *= z;
        }
    }
    quantum_decohere(reg);
}

```

Fig. 2. Example of two functions from the benchmark `libquantum` with different CFGs that could be merged, as shown at the bottom. We highlight where they differ.

responsible for arranging the code in segments that are either equivalent or non-equivalent. We implement this technique at the level of the intermediate representation (IR). Our current implementation assumes that the input functions have all their ϕ -functions demoted to memory operations, simplifying our code generation.

The proposed technique consists of three major steps, as depicted in Figure 3. First, we linearize each function, representing the CFG as a sequence of labels and instructions. The second step consists of applying a sequence alignment algorithm, borrowed from bioinformatics, which identifies regions of similarity between sequences. The sequence alignment algorithm allows us to arrange two linearized functions into segments that are equivalent between the two functions and segments where they differ from one another. The final

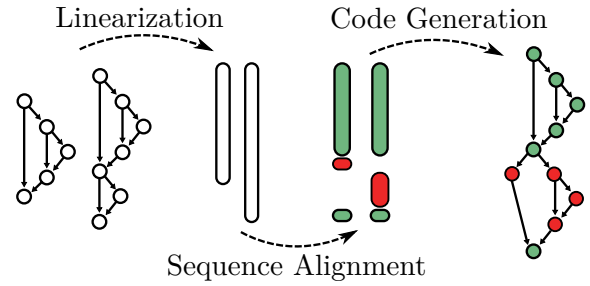


Fig. 3. Overview of our function-merging technique. Equivalent segments of code is represented in light green and the non-equivalent ones in dark red.

step performs the code generation, actually merging the two functions. Aligned segments with equivalent code are merged, avoiding redundancy, while the remaining segments where the two functions differ have their code guarded by a function identifier.

During code generation, we create a merged list of parameters, including the extra function identifier if there are any dissimilar segments. Arguments of the same type are shared between the functions, without necessarily keeping their original order. We also allow for the return types to be different, in which case we use an aggregate type to return values of both types. If one of them is void, then we do not create an aggregate type, we just return the non-void type. Given the appropriate function identifier, the merged function is semantically equivalent to the original functions, so we replace all of their invocations with the new function. It should be noted that in the special case where we merge identical functions, the output is also identical, emulating the behavior of function merging in production compilers.

After producing the merged function, the bodies of the original functions are replaced by a single call to this new function, creating what is sometimes called a *thunk*. In some cases, it may also be valid and profitable to completely delete the original functions, remapping all their original calls to the merged function. Two of the key facts that prohibit the complete removal of the original functions are the existence of indirect calls or the possibility of external linkage.

B. Linearization

Linearization² is a key step for enabling the use of sequence alignment. It takes the CFG of the function, specifies a traversal order of the basic blocks, and for each block outputs its label and its instructions. Linearization maintains the original ordering of the instructions inside each basic block. The edges of the CFG are implicitly represented with branch instructions having the target labels as operands. Figure 4 shows a simplified example of linearizing the CFG of a real function extracted from the SPEC CPU2006 400.perlbench benchmark.

The traversal order we use for linearization has no effect on the correctness of the transformation but it can impact its effectiveness. We empirically chose a reverse post-order

²Although linearization of CFGs usually refers to a predicated representation, in this paper, we use a simpler definition.

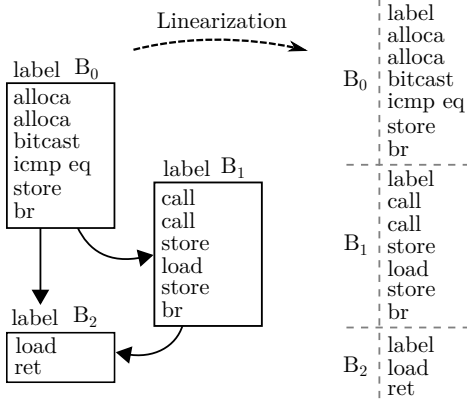


Fig. 4. Linearizing the CFG of an example function.

traversal with a canonical ordering of successor basic blocks. This strategy leads to good performance in our experiments.

C. Sequence Alignment

When merging two functions, the goal is to identify which segments of the code are equivalent (and therefore can be merged) and which ones are different. To avoid breaking the semantics of the original program, we also need to maintain the order of the instructions for each of the functions.

After linearization, we reduce the problem of merging functions to the problem of *sequence alignment*. Sequence alignment is important in many scientific areas, most notably in molecular biology [14], [15], [16], [17] where it is used for identifying homologous subsequences of amino acid in proteins. Figure 5 shows an example of the sequence alignment between two linearized functions extracted from the `400.perlbench` benchmark, including the one used in Figure 4. Essentially, sequence alignment algorithms insert blank characters in both input sequences so that the final sequences end up having the same size, where equivalent segments are aligned with their matching segments from the other sequence and non-equivalent segments are paired with blank characters.

Formally, sequence alignment can be defined as follows: For a given alphabet α , a sequence S of k characters is an element of α^k , i.e., $S = (a_1, \dots, a_k)$. Let S_1, \dots, S_m be a set of sequences, possibly of different lengths but all derived from the same alphabet α , where $S_i = (a_1^{(i)}, \dots, a_{k_i}^{(i)})$, for all $i \in \{1, \dots, m\}$. Consider an extended alphabet that includes the *blank* character “-”, i.e., $\beta = \alpha \cup \{-\}$. An alignment of the m sequences, S_1, \dots, S_m , is another set of sequences, $\bar{S}_1, \dots, \bar{S}_m$, such that each sequence \bar{S}_i is obtained from S_i by inserting blanks in positions where some of the other sequences have non-blank and possibly equivalent characters, for a given equivalence relation. All sequences \bar{S}_i in the alignment set have the same length l , where $\max\{k_1, \dots, k_m\} \leq l \leq k_1 + \dots + k_m$. Moreover, $\forall i \in \{1, \dots, m\}$, $\bar{S}_i = (b_1^{(i)}, \dots, b_l^{(i)})$, there are increasing functions $v_i : \{1, \dots, k_i\} \rightarrow \{1, \dots, l\}$, such that:

- $b_{v_i(j)}^{(i)} = a_j^{(i)}$, for every $j \in \{1, \dots, k_i\}$;

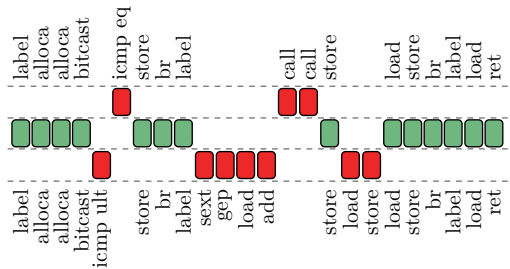


Fig. 5. The sequence alignment between two functions, identifying the equivalent segments of code (green in the center) and the non-equivalent ones (red at the sides).

- any position j not mapped by the function v_i , i.e., for all $j \in \{1, \dots, l\} \setminus \text{Im } v_i$, then $b_j^{(i)}$ is a blank character.

Finally, for all $j \in \{1, \dots, l\}$, there is at least one value of i for which $b_j^{(i)}$ is not a blank character. Note that two aligned sequences may contain both non-blank and non-equivalent characters at any given position, in which case there is a mismatch.

Specifically for function-merging, we are concerned with the alphabet consisting of all possible typed instructions and labels. Every linearized function represents a sequence derived from this alphabet. We explain the equivalence relation used for this alphabet in the next section. Although we only consider pair-wise alignments, the technique would also work for multi-sequences.

Our work uses the Needleman-Wunsch algorithm [14] to perform sequence alignment. This algorithm gives an alignment that is guaranteed to be optimal for a given scoring scheme [18], however, other algorithms could also be used with different performance and memory usage trade-offs [14], [15], [16], [19]. Different alignments would produce different but valid merged functions.

The Needleman-Wunsch algorithm [14] is based on dynamic programming and consists of two main steps. First, it builds a *similarity matrix*, based on a scoring scheme, which assigns weights for matches, mismatches, and *gaps* (blank characters). Afterwards, a backward traversal is performed on the similarity matrix, in order to reconstruct the final alignment by maximizing the total score. We use a standard scoring scheme for the Needleman-Wunsch algorithm that rewards matches and equally penalizes mismatches and gaps.

D. Equivalence Evaluation

Before we merge functions, we first need to define what makes two pieces of code equivalent and therefore mergeable. In this section, we define equivalence in two separate cases, the equivalence between instructions and the equivalence between labels.

In general, two instructions are equivalent if: (1) their opcodes are semantically equivalent, but not necessarily the same; (2) they both have equivalent types; and (3) they have pairwise operands with equivalent types. Types are equivalent if they can be bitcast in a lossless way from one to the other. For pointers, we also need to make sure that there is no conflict regarding memory alignment. In the special case of

function calls, type equivalence means that both instructions have identical function types, i.e. identical return types and identical list of parameters.

Labels can represent both normal basic blocks and landing blocks used in exception handling code. Labels of normal basic blocks are ignored during code equivalence evaluation, but we cannot do the same for landing blocks. We describe how we handle such blocks in more detail in the following section.

Exception Handling Code: Most modern compilers, including GCC and LLVM, implement the zero-cost Itanium ABI for exception handling [20] sometimes called the *landing-pad* model. This model consists of: (1) invoke instructions that have two successors, one for the normal execution and one for handling exceptions, called the landing block; (2) landing-pad instructions that encode which action is taken when an exception has been thrown. The invoke instruction co-operates tightly with its landing block. The landing block must have a landing-pad instruction as its first non- ϕ instruction. As a result, two equivalent invoke instructions must also have landing blocks with identical landing-pad instructions. This verification is made easy by having the landing-pad instruction as the first instruction in a landing block. Similarly, landing-pad instructions are equivalent if they have exactly the same type and also encode identical lists of exception and cleanup handlers.

E. Code Generation

The code generation phase is responsible for producing a new function from the output of the sequence alignment. Our four main objectives are: merging the parameter lists; merging the return types; generating select instructions to choose the appropriate operands in merged instructions; and constructing the CFG of the merged function.

Our approach can effectively handle multiple different function merging scenarios:

- identical functions,
- functions with differing bodies,
- functions with different parameter lists,
- functions with different return types,
- and any combination of these cases.

To maintain the semantics of the original functions, we must be able to pass their parameters to the new merged function. The merged parameter list is the union of the original lists, with placeholders of the correct type for any of the parameters. Maintaining the original order is not important for maintaining semantics, so we make no effort to do so. If the two functions have differing bodies, we add an extra binary parameter, called the function identifier, to the merged list of parameters. This extra parameter is required for selecting code that should be executed only for one of the merged functions.

Figure 6 depicts how we merge the list of parameters of two functions. First, we create the binary parameter that represents the function identifier, one of the functions will be identified by the value `true` and the other by the value `false`. We then add all the parameters of one of the functions to the new list of

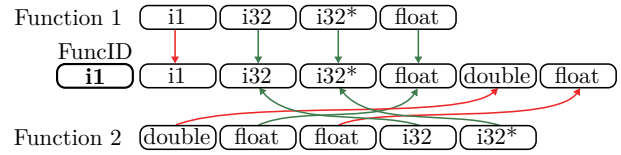


Fig. 6. Example of a merge operation on the parameter lists of two functions.

parameters. Finally, for each parameter of the second function, we either reuse an existing and available parameter of identical type from the first function or we add a new parameter. We keep track of the mapping between the lists of parameters of the original functions and the merged function so that, later, we are able to update the function calls. When replacing the function calls to the new merged function, parameters that are not used by the original function being called will receive undefined values.

The reuse of parameters between the two merged functions provides the following benefits: (1) it reduces the overheads associated with function call abstractions, such as reducing the number of values required to be communicated between functions. (2) if both functions use merged parameters in similar ways, it will remove some of the cases where we need select instructions to distinguish between the functions.

There are multiple valid ways of merging parameter lists. For example, multiple parameters of one function may have the same type as a given parameter from the other function. In such cases, we select parameter pairs that minimize the number of select instructions. We find them by analyzing all pairs of equivalent instruction that use the parameters as operands. Our experiments show that maximizing the matching of parameters, compared to never merging them, improves code-size reduction of individual benchmarks by up to 7%.

Our technique is able to merge any return types. When merging return types, we select the largest one as the base type. Then, we use bitcast instructions to convert between the types. Before a return instruction, we bitcast the values to the base return type. We reverse this at the call-site, where we cast back to the original type. Having identical types or void return are just special cases where casting is unnecessary. In the case of void types, we can just return *undefined* values since they will be discarded at the corresponding call-sites.

After generating the merged list of parameters and return type, we produce the CFG of the merged function in two passes over the aligned sequence. The first pass creates the basic blocks and instructions. The second assigns the correct operands to the instructions and connects the basic blocks. A two-passes approach is required in order to handle loops, due to cyclic data dependencies.

First, for each entry in the aligned sequence, we either create a new basic block for labels or we add a cloned instruction to the appropriate basic block. If the label represents a landing block, a landing-pad instruction is also added to the new basic block. During this process, we keep a mapping from the instructions and labels in the original functions to their corresponding values in the new merged function. We need this mapping to generate the use-definition chains for the

merged function, which is done by pointing the operands of the instructions to the correct values in the function. However, at this point, the cloned instructions are given empty operands, as we are still creating the complete mapping.

While iterating over the aligned sequence, we also need to create extra basic blocks and branch instructions in order to maintain the semantics of the original functions, guarding the execution of instructions that are unique to one of the functions being merged. When transitioning from matching instructions or labels to non-matching ones, we need to branch to new basic blocks based on the function identifier. When transitioning back from non-matching segments to a matching segment, we need to reconnect both divergent points by branching back to a single new basic block where merged instructions will be added. This process generates diamond shaped structures in the CFG.

The second pass over the aligned sequence creates the operands of all instructions. We use the previously created mapping in order to identify the correct operands for each instruction in the merged function. There are two main cases: (1) Creating the operands for non-matching instructions (i.e. those that occur in just one function) is straightforward. In this case, we only need to use the values on which the operands of the original instruction map. (2) Matching instructions can have different values in corresponding operands in each one of the original functions. If this is the case and the original operands map to different values V_1 and V_2 , then we need to choose at runtime the correct value based on the function identifier. We do with an extra select instruction “select (func_id==1), V_1 , V_2 ”, which computes the operand of the merged instruction. If the two values are statically identical, then we do not need a select.

If the operands are labels, instead of adding a select instruction, we perform operand selection through divergent control flow, using a new basic block and a conditional branch on the function identifier. If the two labels represent landing blocks, we hoist the landing-pad instruction to the new common basic block, converting it to a landing block and converting the two landing blocks to normal basic blocks. This is required for the correctness of the landing-pad model.

Similar to previous work on vectorization [21], we also exploit commutative instructions in order to maximize similarity. When assigning operands to commutative instructions, we perform operand reordering to maximize the number of matching operands and reduce the total number of select instructions required. It is also important to note that if we are merging two identical functions, no select or extra branch instruction will be added. As a result, we can remove the extra parameter that represents the function identifier.

IV. FOCUSING ON PROFITABLE FUNCTIONS

Although the proposed technique is able to merge any two functions, it is not always profitable to merge them. In fact, as it is only profitable to merge functions that are sufficiently similar, for most pairs of functions, merging them increases code size. In this section, we introduce our framework for

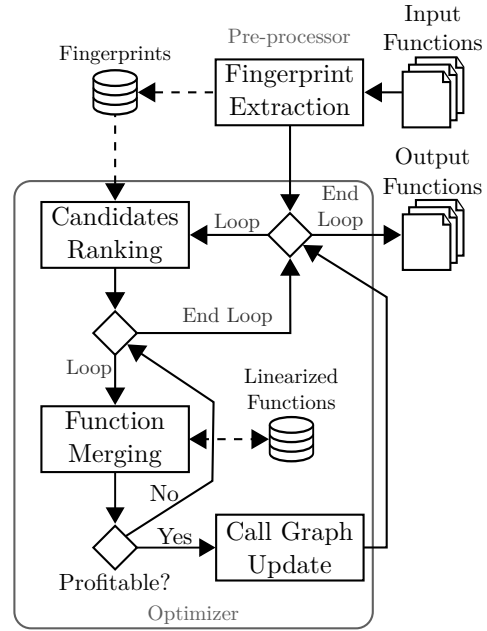


Fig. 7. Overview of our exploration framework.

efficiently exploring the optimization space, focusing on pairs of functions that are profitable to merge.

For every function, ideally, we would like to try to merge it with all other functions and choose the pair that maximizes the reduction in code size. However, this quadratic exploration over all pairs of functions results in prohibitively expensive compilation overhead. In order to avoid the quadratic exploration of all possible merges, we propose the exploration framework shown in Figure 7 for our optimization.

The proposed framework is based on a light-weight ranking infrastructure that uses a *fingerprint* of the functions to evaluate their similarity. It starts by precomputing and caching fingerprints for all functions. The purpose of fingerprints is to make it easy to discard unpromising pairs of functions so that we perform the more expensive evaluation only on the most promising pairs. To this end, the fingerprint consists of: (1) a map of instruction opcodes to their frequency in the function; (2) the set of types manipulated by the function. While functions can have several thousands of instructions, an IR usually has just a few tens of opcodes, e.g., the LLVM IR has only about 64 different opcodes. This means that the fingerprint needs to store just a small integer array of the opcode frequencies and a set of types, which allows for an efficient similarity comparison.

By comparing the opcode frequencies of two functions, we are able to estimate the best case merge, which would happen if all instructions with the same opcode could match. This is a very optimistic estimation. It would be possible only if instruction types and order did not matter. We refine it further by estimating another best case merge, this time based on type frequencies, which would happen if all instructions with the same data type could match.

Therefore, the upper-bound reduction, computed as a ratio,

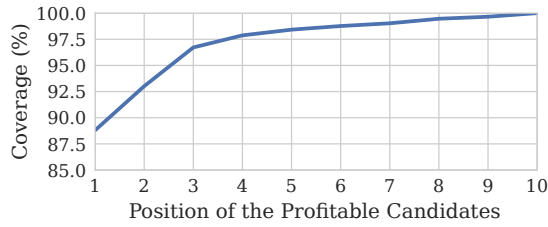


Fig. 8. Average CDF for the position of the profitable candidate and the percentage of merged operations covered. 89% of the merge operations happen with the topmost candidate.

can be generally defined as

$$UB(f_1, f_2, K) = \frac{\sum_{k \in K} \min\{freq(k, f_1), freq(k, f_2)\}}{\sum_{k \in K} freq(k, f_1) + freq(k, f_2)}$$

where $UB(f_1, f_2, Opcodes)$ computes the opcode-based upper bound and $UB(f_1, f_2, Types)$ computes the type-based upper bound. The final estimate selects the minimum upper bound between the two, i.e.,

$$s(f_1, f_2) = \min\{UB(f_1, f_2, Opcodes), UB(f_1, f_2, Types)\}$$

This estimate results in a value in the range $[0, 0.5]$, which encodes a description that favors maximizing both the opcode and type similarities, while also minimizing their respective differences. Identical functions will always result in the maximum value of 0.5.

For each function f_1 , we use a priority queue to rank the topmost similar candidates based on their similarity, defined by $s(f_1, f_2)$, for all other functions f_2 . We use an exploration threshold to limit how many top candidates we will evaluate for any given function. We then perform this candidate exploration in a greedy fashion, terminating after finding the first candidate that results in a profitable merge and committing that merge operation.

Ideally, profitable candidates will be as close to the top of the rank as possible. Figure 8 shows the cumulative distribution of the position of the profitable candidates in a top 10 rank. It shows that about 89% of the merge operations occurred with the topmost candidate, while the top 5 cover over 98% of the profitable candidates. These results suggest that fingerprint similarity is able to accurately capture the real function similarity, while reducing the exploration cost by orders of magnitudes, depending on the actual number and size of the functions.

When a profitable candidate is found, we first replace the body of the two original functions to a single call to the merged function. Afterwards, if the original functions can be completely removed, we update the call graph, replacing the calls to the original functions by calls to the merged function. Finally, the new function is added to the optimization working list. Because of this feedback loop, merge operations can also be performed on functions that resulted from previous merge operations.

A. Profitability Cost Model

After generating the code of the merged function, we need to estimate the code-size benefit of replacing the original pair of functions by the new merged function. In order to estimate the code-size benefit, we first compute the code-size cost for each instruction in all three functions. In addition to measuring the difference in size of the merged function, we also need to take into account all extra costs involved: (1) for the cases where we need to keep the original functions with a call to the merged function; and (2) for the cases where we update the call graph, there might be an extra cost with a call to the merged function due to the increased number of arguments.

Let $c(f)$ be the code-size cost of a given function f , and $\delta(f_i, f_j)$ represent the extra costs involved when replacing or updating function f_i with the function f_j . Therefore, given a pair of functions $\{f_1, f_2\}$ and the merged function $f_{1,2}$, we want to maximize the profit defined as:

$$\Delta(\{f_1, f_2\}, f_{1,2}) = (c(f_1) + c(f_2)) - (c(f_{1,2}) + \varepsilon)$$

where $\varepsilon = \delta(f_1, f_{1,2}) + \delta(f_2, f_{1,2})$. We consider that the merge operation is profitable if $\Delta(\{f_1, f_2\}, f_{1,2}) > 0$.

However, because we are operating on the IR level, one IR instruction does not necessarily translate to one machine instruction. Because of that, the profitability is measured with the help of the compiler’s target-specific cost model. The actual cost of each instruction comes from querying this compiler’s built-in cost model, which provides a target-dependent cost estimation that approximates the code-size cost of an IR instruction when lowered to machine instructions. Our implementation makes use of the code-size costs provided by LLVM’s target-transformation interface (TTI), which is widely used in the decision making of most optimizations [22], [23].

B. Link-Time Optimization

There are different ways of applying this optimization, with different trade-offs. We can apply our optimization on a per compilation-unit basis, which usually results in lower compilation-time overheads because only a small part of the whole program is being considered at each moment. However, this also limits the optimization opportunities, since only pairs of functions within the same translation unit would be merged.

On the other hand, our optimization can also be applied in the whole program, for example, during link-time optimization (LTO). Optimizing the whole program is beneficial for the simple fact that the optimization will have more functions at its disposal. It allows us to merge functions across modules.

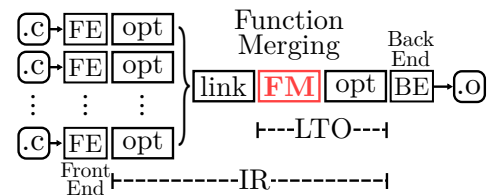


Fig. 9. In our experiments we use a compilation pipeline with a monolithic link-time optimization (LTO).

In addition to the benefit of being able to merge more functions, when optimizing the whole program, we can also be more aggressive when removing the original functions, since we know that there will be no external reference to them. However, if the optimization is applied per translation unit, then extra conditions must be guaranteed, e.g., the function must be explicitly defined as internal or private to the translation unit.

Figure 9 shows an overview of the compilation pipeline used throughout our evaluation. First, we apply early code-size optimizations ($-Os$) to each compilation unit. Then, function merging and further code-size optimizations are applied during monolithic link-time optimization (LTO). With LTO, object file generation is delayed until all input modules are known, instead of being generated per translation unit, which enables more powerful optimizations based on whole-program analyses.

V. EVALUATION

In this section, we evaluate the proposed optimization, where we analyze our improvements on code size reduction, as well as its impact on the program’s performance and compilation-time.

A. Experimental Setup

We compare our optimization against the state-of-the-art [13] and LLVM’s identical [12] function merging techniques. In our evaluation, we refer to the identical function merging as *Identical*, the state-of-the-art as *SOA*, and our approach as *FMSA*. We also run LLVM’s identical function merging before both *SOA* and *FMSA*, as this helps to reduce compilation time by efficiently reducing the number of trivially mergeable functions.

All optimizations are implemented in LLVM v8 and evaluated on two benchmark suites: the C/C++ SPEC CPU2006 [24] and MiBench [25]. We target two different instruction sets, the Intel x86-64 and the ARM Thumb. Our Intel test bench has a quad-core 3.4 GHz Intel Core i7 CPU with 16 GiB of RAM. The ARM test bench has a Cortex-A53 ARMv8 CPU of 1.4 GHz with 1 GiB of RAM. We use the Intel platform for compiling for either target. As a result, compilation-time is almost identical for both targets. Changing the target only affects the behavior of the backend, a very short part of the pipeline. Because of that, we only report compilation-time overhead results for one of the targets, the Intel ISA.

For the proposed optimization, we vary the exploration threshold (Section IV) and we present the results for a range of threshold values. We also show the results for the oracle exploration strategy, which instead of using a rank-based greedy approach, merges a function with all candidates and chooses the best one. This oracle is a perfect ranking strategy but is unrealistic. It requires a very costly quadratic exploration, as explained in Section IV.

B. Code-Size Reduction

Figure 10 reports the code size reduction over the baseline for the linked object. We observe similar trends of code size

reduction on both target architectures. This is expected because the optimizations are applied at the platform-independent IR level. Changing the target architecture introduces only second order effects, such as slightly different decisions due to the different cost model (LLVM’s TTI) and differences in how the IR is encoded into binary.

Our approach, FMSA, significantly improves over the state-of-the-art (SOA). For the Intel platform, FMSA can achieve an average code size reduction of up to 6.3% (or 6% with the lowest exploration threshold), while the SOA and Identical had an average reduction of 2.5% and 1.4%, respectively. Similarly, for the ARM platform, FMSA can achieve an average code size reduction of up to 6.1% (or 5.7% with the lowest threshold), while SOA and Identical had an average reduction of 3% and 1.8%, respectively. For several of the benchmarks, the proposed technique achieves impressive code size reduction compared to other merging approaches.

In most cases, LLVM’s identical function merging has very little impact on code size. We see noticeable impact only on some of the C++ benchmarks, namely, 447.dealII, 450.soplex, 471.omnetpp, 483.xalancbmk. These are the cases that identical function merging was designed to handle, duplicate functions due to heavy use of templating. Although the state-of-the-art improves over LLVM’s identical function merging, it still gets most of its code size reduction for benchmarks with heavy use of templating. In addition to achieving better results in all of these cases, our technique also shows remarkable reductions on several of the C benchmarks, especially 462.libquantum and 482.sphinx3, where other techniques have no real impact.

In Section II, we show two examples extracted from 462.libquantum and 482.sphinx3, where we detail how existing techniques fail to merge similar functions in these benchmarks. Our technique is the *first* that can handle these examples, producing merged functions equivalent to the handwritten ones shown in Figures 1 and 2.

TABLE I
NUMBER AND SIZE OF FUNCTIONS PRESENT IN EACH SPEC CPU2006 BENCHMARK JUST BEFORE FUNCTION MERGING, AS WELL AS NUMBER OF MERGE OPERATIONS APPLIED BY EACH TECHNIQUE.

Benchmarks	#Fns	Min/Avg/Max Size	Identical	SOA	FMSA[t=1]	FMSA[t=10]
400.perlbenc	1699	1 / 125 / 12501	12	103	175	200
401.bzip2	74	1 / 206 / 5997	0	0	7	7
403.gcc	4541	1 / 127.7 / 20688	136	341	614	710
429.mcf	24	18 / 87.25 / 297	0	1	1	1
433.milc	235	1 / 67.69 / 416	0	6	26	34
444.namd	99	1 / 570.64 / 1698	1	1	5	5
445.gobmk	2511	1 / 43.22 / 3140	183	485	436	605
447.dealII	7380	1 / 60.63 / 4856	1835	2785	2974	3315
450.soplex	1035	1 / 73.27 / 1719	27	125	156	163
453.povray	1585	1 / 98.05 / 5324	60	112	193	212
456.hmmer	487	1 / 99.98 / 1511	3	16	45	47
458.sjeng	134	1 / 145.06 / 1252	0	5	11	11
462.libquantum	95	1 / 56.64 / 626	0	1	9	9
464.h264ref	523	1 / 171.42 / 5445	3	22	50	52
470.libm	17	6 / 123.41 / 680	0	0	0	0
471.omnetpp	1406	1 / 26.9 / 611	45	69	227	270
473.astar	101	1 / 67.11 / 584	0	2	4	4
482.sphinx3	326	1 / 80 / 924	2	6	24	26
483.xalancbmk	14191	1 / 38.58 / 3809	3057	4573	4342	4887

Table I provides detailed statistics for the SPEC CPU2006. We show how many functions (#Fns) are present in the linked program just before the merging pass, as well as the average,

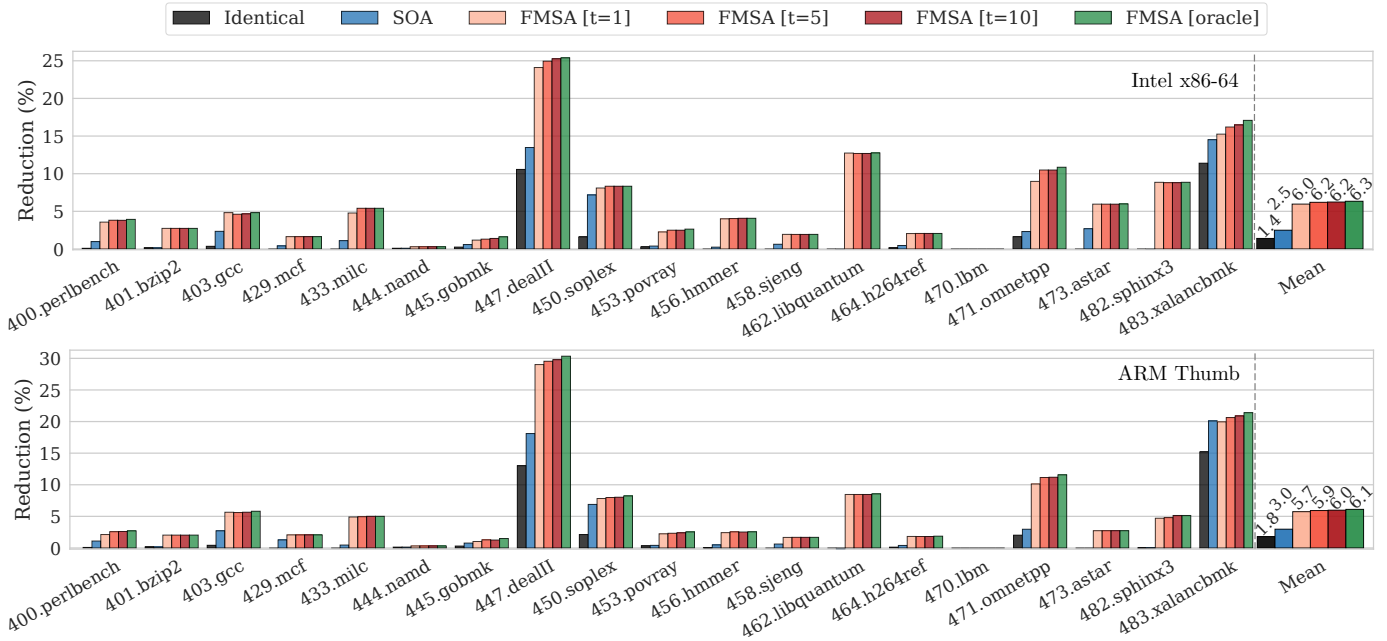


Fig. 10. Object file size reduction for Intel (top) and ARM (bottom). We evaluate our approach (FMSA) under four different exploration thresholds, which control how many potential merging pairs we examine for each function before making a decision. Even for a threshold of one, we outperform the state-of-the-art by $2.4\times$ (Intel) and $1.9\times$ (ARM).

minimum, and maximum size of these functions, in number of instructions, at this same point in the compilation pipeline. We also report how many pair-wise merge operations are performed by each one of the function merging techniques. Note that in almost all cases FMSA performs significantly more merge operations than the other techniques. There are only two cases where FMSA with exploration threshold of one finds fewer profitable merges than the state-of-the-art. This is due to our aggressive pruning of the search space with our ranking mechanism. Simply increasing the threshold, e.g. to ten, allows FMSA to merge more functions. In any case, these extra merge operations of the state-of-the-art have little effect on the overall code size reduction. The state-of-the-art is more likely to fail to merge large functions and succeed with small ones, so even when merging more functions, it does not reduce code size as much as FMSA.

MiBench: Embedded Benchmark Suite: We have shown that our technique achieves good results when applied on the SPEC CPU suite. It reduces size not only on templated C++ benchmarks, like other techniques, but also on C benchmarks where merging opportunities should be almost non-existent. Here, we further explore how FMSA handles such cases by applying it on the MiBench suite, a collection of small C programs each one composed of a small number of functions.

Figure 11 shows the object file reduction for the MiBench programs on the Intel platform. Our best result is for the `rijndael` benchmark, which implements the well-known AES encryption. FMSA merges the two largest functions, namely, `encrypt` and `decrypt`. Inspecting the LLVM IR for the `rijndael` benchmark, we observe that the two functions contain a total of 2494 instructions, over 70% of the code. When we merge them by sequence alignment, we create

a single function with only 1445 instructions, a 42% reduction in the number of IR instructions. This translates into a 20.6% reduction in the linked object file.

TABLE II
NUMBER AND SIZE OF FUNCTIONS PRESENT IN EACH MiBENCH BENCHMARK JUST BEFORE FUNCTION MERGING, AS WELL AS NUMBER OF MERGE OPERATIONS APPLIED BY EACH TECHNIQUE.

Benchmarks	#Fns	Min/Avg/Max Size	Identical	SOA	FMSA[t=1]	FMSA[t=10]
CRC32	4	8 / 24.75 / 39	0	0	0	0
FFT	7	7 / 49.86 / 144	0	0	0	0
adpcm_c	3	37 / 73 / 100	0	0	0	0
adpcm_d	3	37 / 73 / 100	0	0	0	0
basicmath	5	4 / 70.8 / 232	0	0	0	0
bitcount	19	4 / 22.26 / 63	0	1	3	3
blowfish_d	8	1 / 245.38 / 824	0	0	0	0
blowfish_e	8	1 / 245.38 / 824	0	0	0	0
jpeg_c	322	1 / 100.52 / 1269	2	6	8	11
jpeg_d	310	1 / 98.93 / 1269	3	6	10	10
dijkstra	6	2 / 33 / 89	0	0	0	0
ghostscript	3446	1 / 54.2 / 4218	53	53	234	250
gsm	69	1 / 97.06 / 737	0	3	8	8
ispell	84	1 / 105.51 / 1082	0	2	5	5
patricia	5	1 / 77 / 167	0	0	0	0
pgp	310	1 / 88.52 / 1845	0	1	10	10
qsort	2	11 / 50 / 89	0	0	0	0
rijndael	7	46 / 472.29 / 1247	0	0	1	1
rsynth	46	1 / 97.28 / 778	0	0	0	0
sha	7	12 / 53.29 / 150	0	0	0	0
stringsearch	10	3 / 47.9 / 99	0	0	1	1
susan	19	15 / 291.84 / 1212	0	0	1	1
typeset	362	1 / 354.47 / 12125	1	4	31	35

Table II provides more detailed statistics for MiBench. LLVM achieves very limited results, reducing `jpeg_c` by just 0.13%, `jpeg_d` by 0.1%, and `ghostscript` by 0.02%, while having no effect on `typeset`. This happens because all the functions merged by LLVM's identical technique are tiny functions relative to the overall size of the program. Most of these functions comprise of just a few IR instructions. For example, in the `typeset` benchmark, while it is able to merge a pair of functions, they only have five instructions. For the

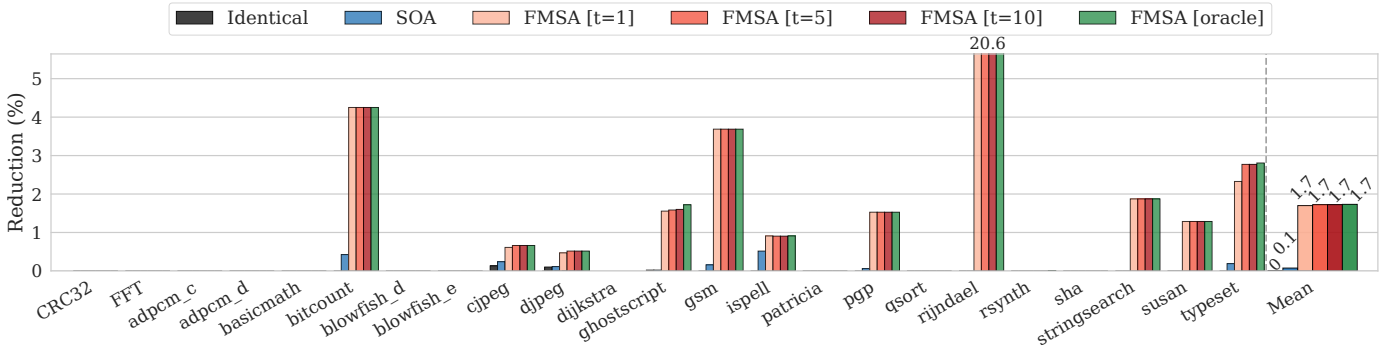


Fig. 11. Object file size reduction for Intel on the Mibench benchmark suite. Our approach (FMSA) is the only one able to achieve a meaningful reduction on these benchmarks.

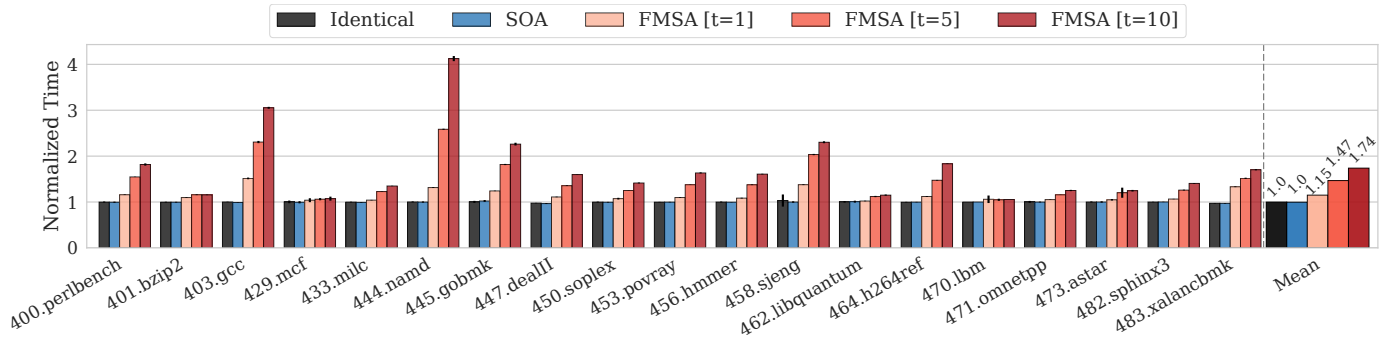


Fig. 12. Compilation-time overhead on the Intel platform. For exhaustive exploration (not shown) the average overhead is $25\times$. Through ranking, we reduce overhead by orders of magnitude. For an exploration threshold of one, FMSA has an overhead of only 20%.

same benchmark, FMSA performs several merge operations, one of them between two functions with over 500 instructions. Overall, the state-of-the-art does slightly better than LLVM’s identical technique but even in its best case it cannot reduce code size more than 0.7%.

Because these embedded benchmarks are much smaller than those in the SPEC suite, trivially similar functions are much less frequent. This is why neither the state-of-the-art nor LLVM’s identical function merging technique had any real effect on these benchmarks. Our technique can look beyond trivially similar functions which allowed it to achieve significant code size reduction on these real embedded benchmarks.

C. Compilation Overhead

Figure 12 shows the compilation-time overhead for all optimizations. As explained in the experimental setup, we only present results when compiling for the Intel platform. Since we cross-compile on the same machine for both targets, compilation times are very similar. We also do not include results for the oracle (exhaustive) exploration. It would be hard to visualize it in the same plot as the other configurations, since it can be up to $136\times$ slower than the baseline.

Unlike the other evaluated techniques, our optimization is a prototype implementation, not yet tuned for compilation-time. We believe that compilation-time can be further reduced with some additional engineering effort. Nevertheless, by using our ranking infrastructure to target only the single most promising equivalent function for each function we examine, we reduce

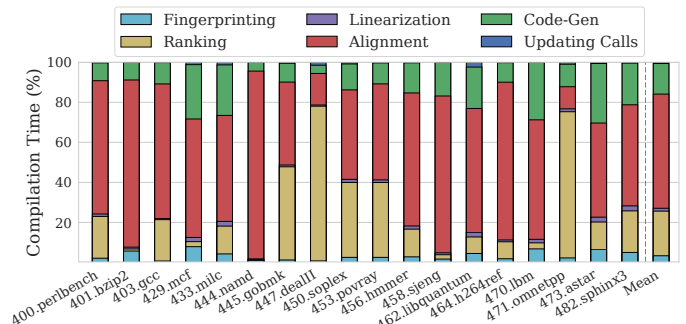


Fig. 13. A compilation-time breakdown isolating the percentage for each major step of the optimization ($t=1$).

compilation-time overhead by up to two orders of magnitude compared to the oracle. This brings the average compile-time overhead to only 20% compared to the baseline, while still reducing code size almost as well as the oracle. Depending on the acceptable trade-off between compilation-time overhead and code size, the developer can change the exploration threshold to exploit more opportunities for code reduction, or to accelerate compilation.

Figure 13 shows a detailed compilation-time breakdown. For each major step of the proposed optimization, we present the accumulated time spent across the whole program. To better understand the overhead of each step, we use an exploration threshold of one ($t = 1$). Because the ranking mechanism performs a quadratic operation on the number of functions,

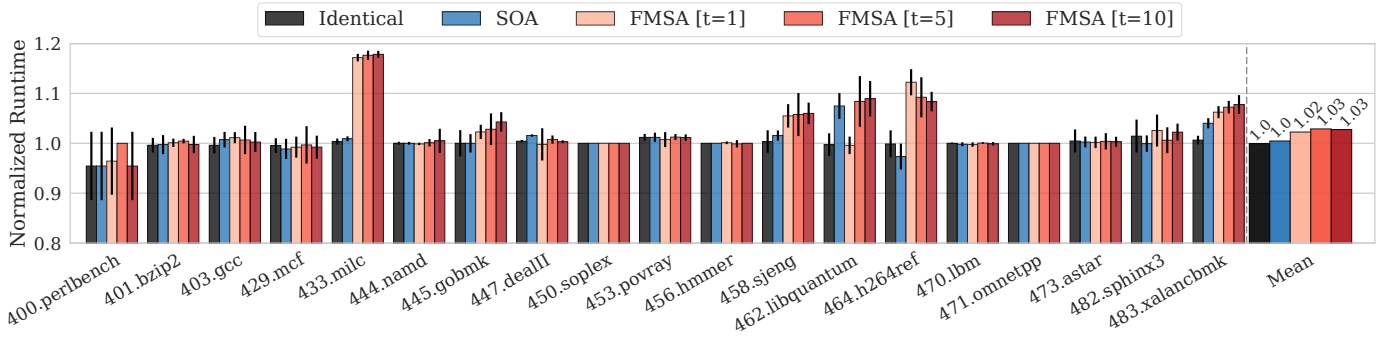


Fig. 14. Runtime overhead on the Intel platform. Performance impact is almost always statistically insignificant. For the few benchmarks affected, FMSA merges hot functions.

computing the similarity between all pairs of functions, it is expected that ranking would be amongst the most costly steps. However, it is interesting to notice that the sequence alignment dominates most of the compilation-time overhead, especially considering that this operation is performed only once per function, when $t = 1$. Although this operation is linear in the number of functions, the Needleman-Wunsh algorithm [14] is quadratic in the size of the functions being aligned, both in time and space. Unsurprisingly, code generation is the third most costly step, which also includes the time to optimize the merge of the parameters. The remaining steps contribute, in total, a small percentage of all the compilation-time overhead. This analysis suggests that optimizing the sequence alignment algorithm and the ranking mechanism is key to reducing even further the overall compilation-time overhead.

D. Performance Impact

The primary goal of function merging is to reduce code size. Nevertheless, it is also important to understand its impact on the programs' execution time and the trade-offs between performance and code size reduction. Figure 14 shows the normalized execution time. Overall, our optimization has an average impact of about 3% on programs' runtime. For most benchmarks, there is no statistically significant difference between the baseline and the optimized binary. Only for 433.milc, 447.dealII, and 464.h264ref there is a noticeable performance impact.

We take 433.milc, which has the worst result, for discussion. For an exploration threshold value of one, we merge 58 functions. Through profiling, we discovered that a handful of them contain hot code, that is, they have basic blocks that are frequently executed. If we prevent these hot functions from merging, all performance impact is removed while still reducing code size. Specifically, our original results showed a 5.11% code size reduction and an 18% performance overhead. Avoiding merging hot functions results in effectively non-existent performance impact and a code size reduction of 2.09%. This code size reduction is still about twice as good as the state-of-the-art. As with the compilation overhead, this is a trade-off that the developer can control.

VI. RELATED WORK

Compiler optimizations for code-size reduction have existed since the very beginning of optimizing compilers. These optimizations can be divided in two main categories: those that replace a piece of code by a smaller but semantically equivalent code, changing the instructions and operations performed [26], [27]; and those that remove or combine redundant code [28], [29], [30], [31], [32], [33], [34]. Function merging falls in the latter category.

A. Function-Merging Techniques

Google developed an optimization for the *gold* linker that merges identical functions on a bit-level [8], [9]. After placing each function in a separate ELF section, they identify functions sections that have their *text* bit-identical and also have their relocations pointing to identical sections. Similar machine-level implementations are also offered by other production compilers and linkers, such as MSVC [35].

This machine-level solution is target-dependent and needs to be adapted for every back-end. A similar optimization for merging identical functions is offered at the IR level by both GCC and LLVM [12], [10]. This optimization is only flexible enough to accommodate simple type mismatches provided they can be bitcast in a lossless way. Its simplicity allows for an efficient exploration approach based on computing the hash of the functions and then using a tree structure to group equivalent functions based on their hash values.

The state-of-the-art function-merging technique exploits structural similarity among functions [13]. Their optimization is able to merge similar functions that are not necessarily identical. Two functions are structurally similar if both their function types are equivalent and their CFGs isomorphic. Two function types are equivalent if they agree in the number, order, and types of their parameters as well as their return types, linkage type, and other compiler-specific properties. In addition to the structural similarity of the functions, their technique also requires that corresponding basic blocks have exactly the same number of instructions and that corresponding instructions must have equivalent resulting types. Mergeable functions are only allowed to differ in corresponding instructions, where they can differ in their opcodes or in the number and type of their input operands.

Because the state-of-the-art is limited to functions with identical CFGs and function types, once it merges a pair of functions, a third *similar* function cannot be merged into the resulting merged function since they will differ in both CFGs and their lists of parameters. Due to this limiting factor, the state-of-the-art has to first collect all mergeable functions and merge them simultaneously.

The state-of-the-art algorithm iterates simultaneously over corresponding basic blocks in the set functions being merged, as they have isomorphic CFGs. For every basic block, if their corresponding instructions have different opcodes, they split the basic block and insert a switch branch to select which instruction to execute depending on a function identifier. Because these instructions have equivalent resulting types, their results can be merged using a phi-operator, which can then be used transparently as operands by other instructions.

Although the state-of-the-art technique improves over LLVM’s identical function merging, it is still unnecessarily limited. In Section II, we showed examples of very similar real functions where the state-of-the-art fails to merge. Our approach addresses such limitations improving on the state-of-the-art across the board.

B. Code Factoring

Code factoring is a related technique that addresses the same fundamental problem of duplicated code in a different way. Code factoring can be applied at different levels of the program [34]. Local factoring, also known as local code motion, moves identical instructions from multiple basic blocks to either their common predecessor or successor, whenever valid [29], [36], [34]. Procedural abstraction finds identical code that can be extracted into a separate function, replacing all replicated occurrences with a function call [34], [37].

Procedural abstraction differs from function merging as it usually works on single basic blocks or single-entry single-exit regions. Moreover, it only works for identical segments of code, and every identical segment of code is extracted into a separate new function. Function merging, on the other hand, works on whole functions, which can be identical or just partially similar, producing a single merged function.

However, all these techniques are orthogonal to the proposed optimization and could complement each other at different stages of the compilation pipeline.

C. Other Applications of Code Similarity

Code similarity has also been used in other compiler optimizations or tools for software development and maintenance. In this section, we describe some of these applications.

Coutinho et al. [38] proposed an optimization that uses instruction alignment to reduce divergent code for GPU. They are able to fuse divergent branches that contain single basic blocks, improving GPU utilization.

Similarly, analogous algorithms have also been suggested to identify the differences between two programs, helping developers with source-code management and maintenance [39],

[40]. These techniques are applied in tools for source-code management, such as the *diff* command [40].

Similar techniques have also been applied to code editors and IDEs [41], [42]. For example, SourcererCC [42] detects possible clones, at the source level, by dividing the programs into a set of code blocks where each code block is itself represented by a bag-of-tokens, i.e., a set of tokens and their frequencies. Tokens are keywords, literals, and identifiers, but not operators. Code blocks are considered clones if their degree of similarity is higher than a given threshold. In order to reduce the number of blocks compared, candidate blocks are filtered based on a few of their tokens where at least one must match.

Our ranking mechanism uses an approach similar to SourcererCC, where we use opcode frequencies and type frequencies to determine if two functions are likely to have similar code. However, we need a precise and effective analysis of code similarity when performing the actual merge. To this end, we use a sequence alignment technique.

VII. CONCLUSION

We introduced a novel technique, based on sequence alignment, for reducing code size by merging arbitrary functions. Our approach does not suffer from any of the major limitations of existing solutions, outperforming them by more than $2.4\times$. We also proposed a ranking-based exploration mechanism to focus the optimization on promising pairs of functions. Ranking reduces the compilation-time overhead by orders of magnitude compared to an exhaustive quadratic exploration. With this framework, our optimization is able to reduce code size by up to 25%, with an overall average of about 6%, while introducing an average compilation-time overhead of only 15%. Coupled with profiling information, our optimization introduces no statistically significant impact on performance.

For future work, we plan to focus on improving the ranking mechanism to reduce compilation time. We envisage further improvements can be achieved by integrating the function-merging optimization to a summary-based link-time optimization framework, such as ThinLTO in LLVM. We also plan to work on the linearization of the candidate functions, allowing instruction reordering to maximize the number of matches between the functions.

APPENDIX

A. Abstract

This artifact provides the source code that implements our function merging optimization as well as the other optimizations required for our evaluation. Our optimization is implemented on top of LLVM v8. We also provide the source code for all benchmarks along with scripts required to reproduce the results presented in the paper. To validate the results build our version of LLVM with the provided scripts, run the benchmarks and, finally, the plotting script to reproduce the main results in the paper.

B. Artifact Check-List

- **Program:** LLVM and Clang, the C/C++ frontend for LLVM; the C/C++ SPEC CPU2006 benchmark suite.
- **Compilation:** With provided scripts.
- **Data set:** Provided with the corresponding benchmarks.
- **Run-time environment:** Linux.
- **Hardware:** Intel architecture.
- **Output:** Raw data in CSV files and plots as PDFs.
- **How much disk space required (approximately)?:** Up to 5 GiB.
- **Publicly available?:** Yes.
- **Workflow frameworks used?:** Download and unzip; build software; run benchmarking scripts; compare output results with the expected plots provided.

C. Description

1) *How Delivered:* The artifact is publicly available. We provide the following option to reproduce our experiments:

Download the source code and benchmark suite, building everything locally on your own machine.
<https://doi.org/10.6084/m9.figshare.7473260>
<https://doi.org/10.6084/m9.figshare.7476149>


The main source file that implements our optimization can be found in the path:

```
llvm/lib/Transforms/IPO/FunctionMerging.cpp
```

The state-of-the-art and LLVM's identical function merging can be found, respectively, in the source files:

```
llvm/lib/Transforms/IPO/MergeSimilarFunctions.cpp  
llvm/lib/Transforms/IPO/MergeFunctions.cpp
```

We provide a detailed demonstration of how to reproduce this artifact in the video at the following URL:

 <http://bit.ly/cgo19fmsa-demo>

2) *Hardware Dependencies:* The experiments described by this artifact were executed on an Intel machine with Intel Core i7-4770 CPU at 3.40 GHz, and 16 GiB of RAM.

3) *Software Dependencies:* In this section, we describe the softwares and packages that must be installed in order to build the LLVM compiler, the benchmark suite, and produce the plots with the results.

The experiments described by this artifact were executed on a machine with the operating system openSUSE Leap 42.2.

Below, we list all Linux and Python packages that must be installed. We also specify the exact version that we have used in our experiments.

- GCC for both C and C++ (gcc, g++)
gcc-4.8-9.61.x86_64
gcc-c++-4.8-9.61.x86_64
binutils-2.29.1-9.6.1.x86_64
- GCC's 32-bits runtime (gcc-multilib, g++-multilib)
gcc-32bit-4.8-9.61.x86_64
gcc-c++-32bit-4.8-9.61.x86_64
- CMake build system (cmake)
cmake-3.5.2-1.2.x86_64
- Python 2.7+ (python)
python-2.7.13-25.3.1.x86_64
- Python's TkInter (python-tk)
python-tk-2.7.13-25.3.1.x86_64
- Python's pip (python-pip)
python-pip-7.1.2-2.4.noarch
- Python's NumPy (numpy)
numpy 1.14.2
- Python's Matplotlib (matplotlib)

```
matplotlib 2.1.0
```

- Python's Seaborn (seaborn)
seaborn 0.9.0

We provide a script, called `setup.sh`, which automatically installs all the necessary dependencies. This script uses the `apt` tool and is the only one that requires `sudo` privileges.

4) *Data Sets:* Datasets are provided as part of the artifact with the benchmark suite.

D. Installation

Download both the source code (<https://doi.org/10.6084/m9.figshare.7473260>) and the benchmark suite (<https://doi.org/10.6084/m9.figshare.7476149>). The source code has a root directory called CGO19FMSA. Unzip all the content comprising the benchmark suite inside the CGO19FMSA root directory. At this point, your CGO19FMSA directory should contain:

```
./CGO19FMSA/  
├── build-all.sh  
├── config.sh  
├── data  
├── expected  
├── llvm  
├── myplots.py  
├── plot-code-size.py  
├── plot-compilation-time.py  
├── plot-execution-time.py  
├── run-all.sh  
├── setup.sh  
└── spec2006
```

In order to install all dependencies described above, run the `setup.sh` script with `sudo` privileges. That is, assuming that you are in the CGO19FMSA directory, run the following command:

```
sudo sh setup.sh
```

Once the dependencies have been installed, run the `build-all.sh` script to build our version of LLVM and Clang, which include our function merging optimization as well as both the state-of-the-art optimization and LLVM's identical function merging optimization. Again, assuming that you are in the CGO19FMSA directory, run the following command:

```
sh build-all.sh
```

This process might take a few hours, depending on your machine settings. After completion, a `build` directory is created inside the CGO19FMSA root directory.

This two scripts set up all the environment necessary to run all our experiments.

E. Experiment Workflow

To run our experiments all you need to do is to execute the `run-all.sh` script with the following command:

```
sh run-all.sh
```

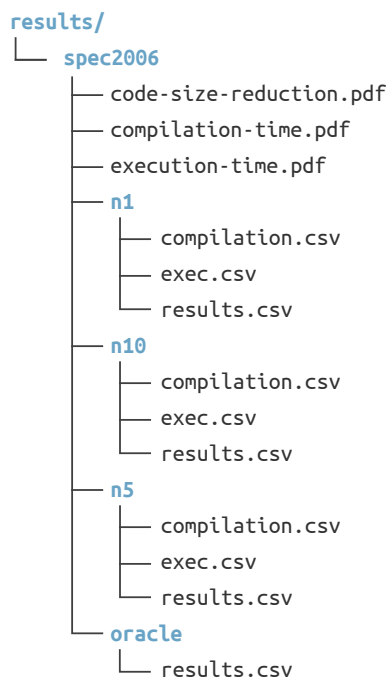
This script automates the whole experiment workflow. At the end, you should have all the expected plots, as well as the raw data as CSV files, inside the `results` directory, which is also created inside the CGO19FMSA root directory.

The automated process may take several hours since it involves running all following steps for all the SPEC benchmarks:

- Code-size measurement (Figure 10 in the paper):
 - Running the oracle optimization.
 - Running the state-of-the-art and baselines optimizations.
 - Running our optimization with all three exploration thresholds.
- Compilation-time measurement (Figure 11 in the paper):
 - Run all optimizations again, except for the oracle, multiple times in order to have a measurement with statistical significance.
- Execution-time measurement (Figure 13 in the paper):
 - Run, also multiple times, all compiled versions of the SPEC benchmarks with their reference inputs.

F. Evaluation and Expected Result

After executing the automated process described above, the `results` directory should have the following content:



The main files to consider are the PDF files, which represent the plots with the main results for the final version of our paper. The three PDF files, as the name of the files suggest, contain the results regarding code-size reduction, compilation-time overhead, and the performance impact of our optimization during execution-time of the benchmarks.

This artifact represents our results for the camera ready version of the paper. We provide the `expected` results that were produced in our environment with the up-to-date version that will be used in the camera ready version of the paper (for example, it includes a benchmark that was missing in the original version of our paper). The reviewers can compare their results with the ones provided in the `expected` directory.

G. Notes

To reduce the overall time required to run the full set of experiments, we have set a small number of repetitions, which may result in large error bars for some of the benchmarks. This effect will depend on the noise of your environment. In order to reduce the noise in the

measurements, you just need to update the `REPEAT` variable in the `CGO19FMISA/run-all.sh` script, changing it to a bigger number.

Because SPEC 2006 requires private access, in the final version, we can provide only our scripts for running the experiments. The source code remains the same (<https://doi.org/10.6084/m9.figshare.7473260>).

ACKNOWLEDGMENT

We would like to thank Tobias Edler von Koch for providing the source code for the state-of-the-art implementation. This work has been supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/L01503X/1 (CDT in Pervasive Parallelism), EP/P003915/1 (SUMMER), EP/M01567X/1 (SANDeRs) and EP/M015793/1 (DIVIDEND). This work was supported by the Royal Academy of Engineering under the Research Fellowship scheme.

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*, M. E. Auer and D. G. Zutin, Eds. Cham: Springer International Publishing, 2018, pp. 19–27.
- [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*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 180–189.
- [3] U. P. Schultz, K. Burggaard, F. G. Christensen, and J. L. Knudsen, "Compiling Java for low-end embedded systems," in *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, ser. LCTES '03. New York, NY, USA: ACM, 2003, pp. 42–50.
- [4] A. Varma and S. S. Bhattacharyya, "Java-through-C compilation: an enabling technology for Java in embedded systems," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 3, Feb 2004, pp. 161–166 Vol.3.
- [5] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder, "Management of resource constrained devices in the internet of things," *IEEE Communications Magazine*, vol. 50, no. 12, pp. 144–149, December 2012.
- [6] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the internet of things: A standardization perspective," *IEEE Internet of Things Journal*, vol. 1, no. 3, pp. 265–275, June 2014.
- [7] 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: Practice and Experience*, vol. 29, no. 22, p. e3932, 2017.
- [8] S. Tallam, C. Coutant, I. L. Taylor, X. D. Li, and C. Demetriou, "Safe ICF: Pointer safe and unwinding aware identical code folding in gold," in *GCC Developers Summit*, 2010.
- [9] D. Kwan, J. Yu, and B. Janakiraman, "Google's C/C++ toolchain for smart handheld devices," in *Proceedings of Technical Program of 2012 VLSI Technology, System and Application*, April 2012, pp. 1–4.
- [10] M. Liška, "Optimizing large applications," *arXiv preprint arXiv:1403.6997*, 2014.
- [11] V. M. Weaver and S. A. McKee, "Code density concerns for new architectures," in *2009 IEEE International Conference on Computer Design*, Oct 2009, pp. 459–464.
- [12] "The LLVM Compiler Infrastructure. MergeFunctions pass, how it works," <http://llvm.org/docs/MergeFunctions.html>.
- [13] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: ACM, 2014, pp. 85–94.
- [14] 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.
- [15] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981.

- [16] H. Carrillo and D. Lipman, "The multiple sequence alignment problem in biology," *SIAM J. Appl. Math.*, vol. 48, no. 5, pp. 1073–1082, Oct. 1988.
- [17] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *Journal of Computational Biology*, vol. 1, no. 4, pp. 337–348, 1994.
- [18] D. G. Higgins and P. M. Sharp, "Fast and sensitive multiple sequence alignments on a microcomputer," *Bioinformatics*, vol. 5, no. 2, pp. 151–153, 1989.
- [19] G. Hickey and M. Blanchette, "A probabilistic model for sequence alignment with context-sensitive indels," in *Proceedings of the 15th Annual International Conference on Research in Computational Molecular Biology*, ser. RECOMB'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 85–103.
- [20] C. de Dinechin, "C++ exception handling," *IEEE Concurrency*, vol. 8, no. 4, pp. 72–79, Oct. 2000.
- [21] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes, "Look-ahead SLP: Auto-vectorization in the presence of commutative operations," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 163–174.
- [22] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes, "VW-SLP: Auto-vectorization with adaptive vector width," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: ACM, 2018, pp. 12:1–12:15.
- [23] A. Pohl, B. Cosenza, and B. Juurlink, "Cost modelling for vectorization on ARM," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2018, pp. 644–645.
- [24] SPEC, "Standard Performance Evaluation Corp Benchmarks," <http://www.spec.org>, 2014.
- [25] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.
- [26] H. Massalin, "Superoptimizer: A look at the smallest program," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS II. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 122–126.
- [27] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson, "Using peephole optimization on intermediate code," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 1, pp. 21–36, Jan. 1982.
- [28] J. Cocke, "Global common subexpression elimination," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: ACM, 1970, pp. 20–24.
- [29] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 147–158.
- [30] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco, "Code compression," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97. New York, NY, USA: ACM, 1997, pp. 358–365.
- [31] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '99. New York, NY, USA: ACM, 1999, pp. 1–9.
- [32] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, Mar. 2000.
- [33] W. K. Chen, B. Li, and R. Gupta, "Code compaction of matching single-entry multiple-exit regions," in *Static Analysis*, R. Cousot, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 401–417.
- [34] G. Lóki, Á. Kiss, J. Jász, and Á. Beszédés, "Code factoring in GCC," in *Proceedings of the 2004 GCC Developers' Summit*, 2004, pp. 79–84.
- [35] "Microsoft Visual Studio. Identical COMDAT folding," <https://msdn.microsoft.com/en-us/library/bxwfs976.aspx>.
- [36] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 159–170.
- [37] A. Dreweke, M. Worlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen, "Graph-based procedural abstraction," in *International Symposium on Code Generation and Optimization (CGO'07)*, March 2007, pp. 259–270.
- [38] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr., "Divergence analysis and optimizations," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011, pp. 320–329.
- [39] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.
- [40] W. Miller and E. W. Myers, "A file comparison program," *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.
- [41] M. Toomim, A. Begel, and S. L. Graham, "Managing duplicated code with linked editing," in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, Sept 2004, pp. 173–180.
- [42] H. Sajjani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Source-ercc: Scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 1157–1168.