Function Merging by Sequence Alignment

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 3× better than the state-of-the-art, reducing code size by up to 22%, with an overall average of 5.3%, while introducing an average compilation-time overhead of only 20%. When aided by profiling information, this optimization can be deployed without any significant impact on the performance of the generated code.

Keywords Code Size, Function Merging, IPO

1. 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 [24]. 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 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 [10].

In such constrained scenarios, reducing the code size is essential [1, 17, 26, 27, 33]. Unfortunately, production compilers offer little help beyond dead-code elimination or merging identical functions [19, 20, 31]. Developers might have more luck just removing functionality from their libraries [17] or hand-optimizing their code [35].

Function merging reduces replicated code by combining multiple identical functions into a single one. Although a simple and intuitive concept, it is crucial for making high-level abstractions usable, when they introduce duplicate code [19, 31]. For example, some C++ ABIs may end up creating multiple identical constructors and destructors of a class to use in different contexts [19] and C++ templates replicate code for different specializations [20, 31]. More advanced approaches [11] 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 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 approach 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, more than three times better than the state-of-the-art can achieve.

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 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 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 avoids this by incorporating profiling information to identify hot code blocks and effectively disable code size optimizations for them.

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.
- We use Sequence Alignment Function Merging to reduce code size by up to 22%, outperforming the state-of-the-

```
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);
}
```

Figure 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.

art by three times, while introducing minimal compile-time and negligible run-time overheads.

2. 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 or 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 [11], 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.

```
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);
}
```

Figure 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.

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.

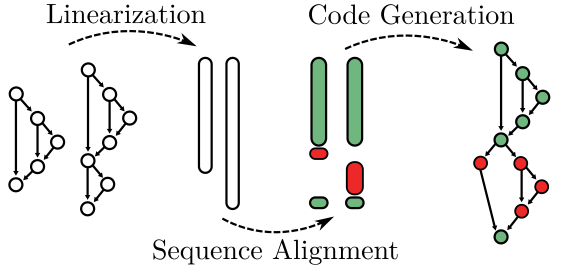

Figure 3. Overview of our function-merging technique. Equivalent segments of code is represented in light green and the non-equivalent ones in dark red.

3. 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 4.1). To avoid an expensive quadratic exploration, we integrate our profitability analysis with an efficient ranking mechanism based on a lightweight fingerprint of the functions.

3.1 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 responsible for arranging the code in segments that are either equivalent or non-equivalent.

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 in 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 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.

At this point, we create a merged list of parameters, including the extra function identifier if there 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.

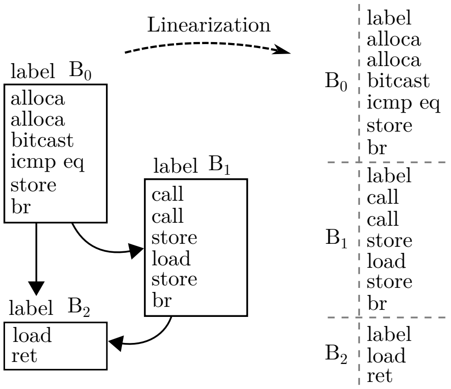

Figure 4. Linearizing the CFG of an example function.

3.2 Linearization

Linearization[1] 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. 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 traversal with a canonical ordering of successor basic blocks. This strategy leads to good performance in our experiments.

3.3 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 one of the functions.

After linearization, we reduce the problem of merging functions to the problem of *sequence alignment*. Sequence alignment is an important technique to many areas of science, most notably in molecular biology [3, 23, 28, 34] where, for example, 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 in a way 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.

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

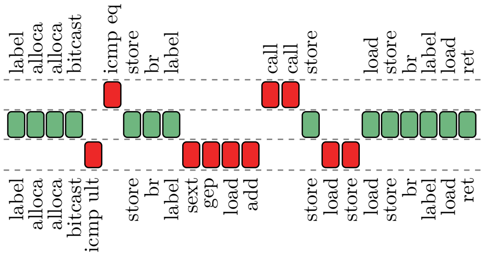

Figure 5. The sequence alignment between two functions, identifying the equivalent segments of code (green in the center) and the non-equivalent ones (red at the sides).

Formally, sequence alignment can be defined as follows: For a given alphabet α, a sequence S of k characters is a subset 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 $S_i = (a_1^{(i)}, \dots, a_{k_1}^{(i)})$, 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 $v_i : \{1, \dots, k_i\} \to \{1, \dots, l\}$, such that:

- $b_{v_i(j)}^{(i)} = a_j^{(i)}$, for every $j \in \{1, \dots, k_i\}$;
- any position j not mapped by the function v_i, i.e., for all $j \in \{1, \dots, l\} \setminus \mathrm{Im} v_i$, 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 it contains 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-Wunsh algorithm [23] to perform sequence alignment. This algorithm gives an alignment that is guaranteed to be optimal for a given scoring scheme [14], however, other algorithm could also be used with different performance and memory usage trade-offs [3, 13, 23, 28]. Different alignments would produce different but valid merged functions.

The Needleman-Wunsh algorithm [23] 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-Wunsh algorithm that rewards matches and equally penalizes mismatches and gaps.

3.4 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.

3.4.1 Exception Handling Code

Most modern compilers implement the zero-cost Itanium ABI for exception handling [7], including GCC and LLVM, 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.

3.5 Code Generation

The code generation phase is responsible for producing a new function from the output of the sequence alignment. Our three main objectives are: merging the parameter lists; 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,

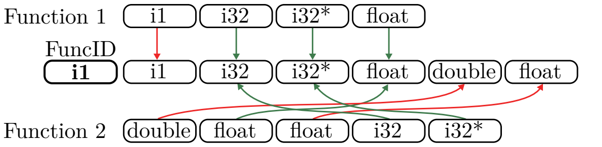

Figure 6. Example of a merge operation on the parameter lists of two functions.

- 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 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%.

After generating the merged list of parameters, 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.

The second pass over the aligned sequence updates the operands of all the instructions and adds select instructions as necessary. Select instructions are created when a merged instruction has different operand values in the original functions, so the appropriate value needs to be selected based on the function identifier. 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 [25], 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 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.

4. 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 propose our framework for efficiently exploring the optimization space, focusing on pairs of functions that are profitable to merge.

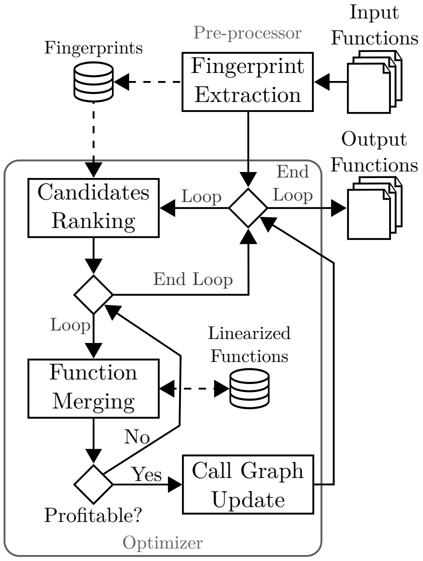

Figure 7. Overview of our exploration framework.

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 first precomputing and caching the fingerprint of all functions. The goal of the fingerprint is to allow us to efficiently discard unpromising pairs of functions so that we perform the more expensive evaluation only on the topmost similar 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 an upper bound of the merge between these functions, assuming that all instructions with the same opcode would always result in a match. This assumption provides an upper bound on the actual number of matches, since it may be affected by the instruction types and the order they appear in the linearized functions. As a way to refine this estimate, we weight this upper bound by the Jaccard similarity coefficient [16] of the sets of types, i.e., a type-similarity ratio between the two functions. Formally, let T_1 and T_2 be the set of types of the functions f_1 and f_2, respectively. Therefore, the upper-bound reduction, computed as a ratio, can be defined as

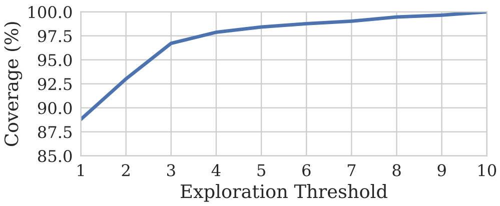

Figure 8. Average CDF for the exploration threshold and the percentage of merged operations covered. 89% of the merge operations happen with the topmost candidate.

$$UB(f_1, f_2) = \frac{\sum\limits_{op \in Ops} \min\{freq(op, f_1), freq(op, f_2)\}}{\sum\limits_{op \in Ops} freq(op, f_1) + freq(op, f_2)}$$

and the weighted estimate is given by

$$s(f_1, f_2) = UB(f_1, f_2) \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|}.$$

This weighted 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. This candidate exploration is then performed in a greedy fashion, where the first candidate that actually results in a profitable merge ends the exploration and the merge operation is finally committed.

Ideally, the profitable candidate 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 the proposed fingerprint similarity is able to accurately capture the real function similarity, while reducing the exploration cost by a few 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.

4.1 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.

4.2 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 compilation 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.

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 compilation unit, then extra conditions must be guaranteed, e.g., the function must be explicitly defined as internal or private to the compilation unit.

Figure 9 shows an overview of the compilation pipeline used throughout our evaluation. First, we apply early code-size optimizations 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 compilation unit, which enables more powerful optimizations based on whole-program analyses.

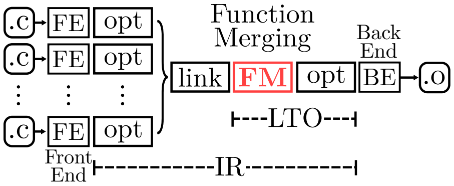

Figure 9. In our experiments we use a compilation pipeline with a monolithic link-time optimization (LTO).

5. 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.

5.1 Experimental Setup

We compare our optimization against the state-of-the-art function merging approach of Edler von Koch et al. [11] and LLVM's identical function merging. In our evaluation, we refer to identical function merging as *Identical*, the state-of-the-art as *SoA*, and our approach as *FMSA*.

All optimizations are implemented in LLVM v7 and evaluated on the C/C++ SPEC CPU2006 [29] benchmark suite. 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 4) 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 4.

5.2 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 5.8% (or 5.3% with the lowest exploration threshold), while the SoA

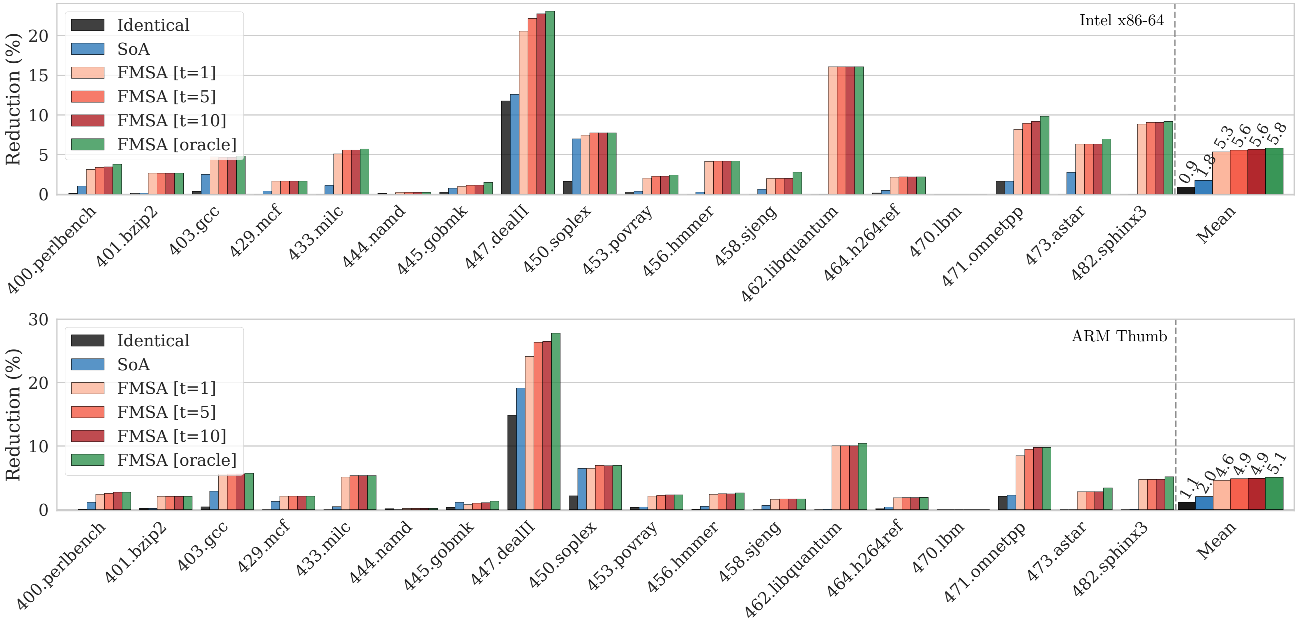

Figure 10. Object file size reduction for Intel (top) and ARM (bottom). We evaluate our approach (FMSA) under four different exploration thresholds, which control how many potential merging pairs we examine for each function before making a decision. Even for a threshold of one, we outperform the state-of-the-art by 2.9×(Intel) and 2.3× (ARM).

and Identical had an average reduction of 1.8% and 0.9%, respectively. Similarly, for the ARM platform, FMSA can achieve an average code size reduction of up to 5.1% (or 4.6% with the lowest threshold), while SoA and Identical had an average reduction of 2% and 1.1%, respectively. For several of the benchmarks, the proposed technique achieves impressive code size reduction compared to other merging approaches. Table 5.2 highlights some of these results for both platforms. In two cases, `462.libquantum` and `482.sphinx3`, the state-of-the-art slightly increases code size, while FMSA reduces size significantly. The only case where SOA outperforms FMSA is on the `445.gobmk` benchmark on the ARM platform. There are little opportunities for function merging on that benchmark and FSMA misses a few of them. However, we can easily fix this by increasing the exploration threshold. For exhaustive exploration, FMSA improves over the SoA, with a code size reduction of 1.33%. In all other cases, FMSA is always equal or better than all other function-merging optimizations, usually by a significant margin.

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`, and `471.omnetpp`. These are the cases that identical function merging was designed to handle, duplicate functions due to heavy use of templating. But even on these benchmarks FMSA outperforms LLVM, with an improvement of almost 6× on `471.omnetpp`. FMSA is designed to merge a superset of the functions that the LLVM identical function merging can handle, so it is able to achieve better results in most of the cases. Moreover, 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.

Benchmarks	Identical	SoA	FMSA [t=1]
401.bzip2	0.15 / 0.16	0.15 / 0.16	2.67 / 2.09
433.milc	0 / 0	1.08 / 0.46	5.09 / 5.12
447.dealII	11.77 / 14.83	12.59 / 19.12	20.57 / 24.08
445.gobmk	0.26 / 0.32	0.77 / 1.15	0.96 / 0.78
453.povray	0.28 / 0.33	0.38 / 0.40	2.04 / 2.12
456.hmmer	0 / 0	0.27 / 0.50	4.15 / 2.40
462.libquantum	0 / 0	-0.07 / -0.17	16.08 / 10.04
471.omnetpp	1.65 / 2.08	1.64 / 2.27	8.18 / 8.47
473.astar	0 / 0	2.76 / 0	6.33 / 2.81
482.sphinx3	0 / 0	-0.06 / 0.06	8.85 / 4.72

Table 1. Highlights of code reduction results (in percentages) on *Intel / ARM* platforms.

5.3 Compilation Overhead

Figure 11 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× 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 compilation-time overhead by up to two

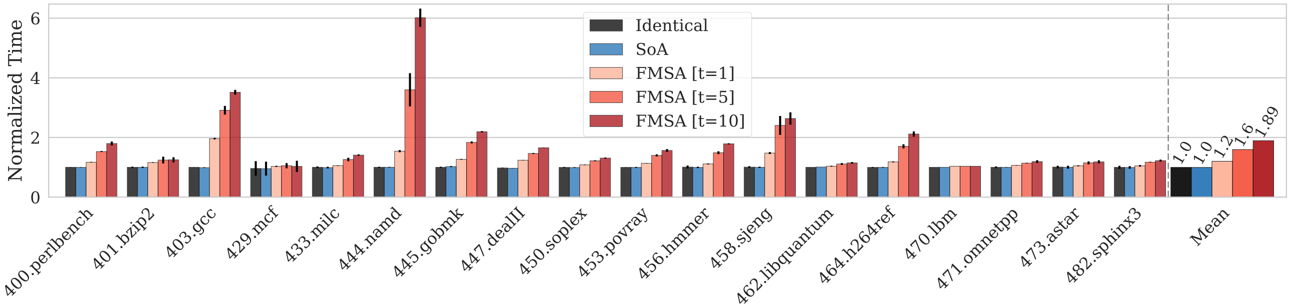

Figure 11. Compilation-time overhead on the Intel platform. For exhaustive exploration (not shown) the average overhead is 25×. Through ranking, we reduce overhead by orders of magnitude. For an exploration threshold of one, FMSA has an overhead of only 20%.

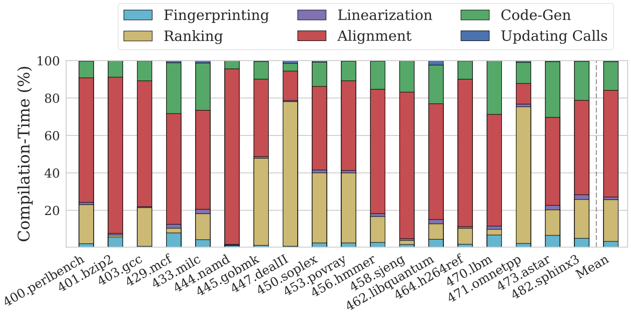

Figure 12. A compilation-time breakdown isolating the percentage for each major step of the optimization (t=1).

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 12 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 one of the steps, we use an exploration threshold of one ($t = 1$). Because the ranking mechanism performs a quadratic operation on the number of 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 on the number of functions, the Needleman-Wunsh algorithm [23] is quadratic on 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.

5.4 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 13 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 for this benchmark. 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. By avoiding merging hot functions, it 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.

6. Related Work

Compiler optimizations for code-size reduction exist 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 [22, 32]; and those that remove or combine redundant code [4–6, 8, 12, 18, 21]. Function merging falls in the latter category.

6.1 Function-Merging Techniques

Google developed an optimization for the *gold* linker that merges identical functions on a bit-level [19, 31]. After placing each function in a separate ELF section, they identify functions sections that have their *text* bit-identical and also their relocations point to identical sections. Similar machine-

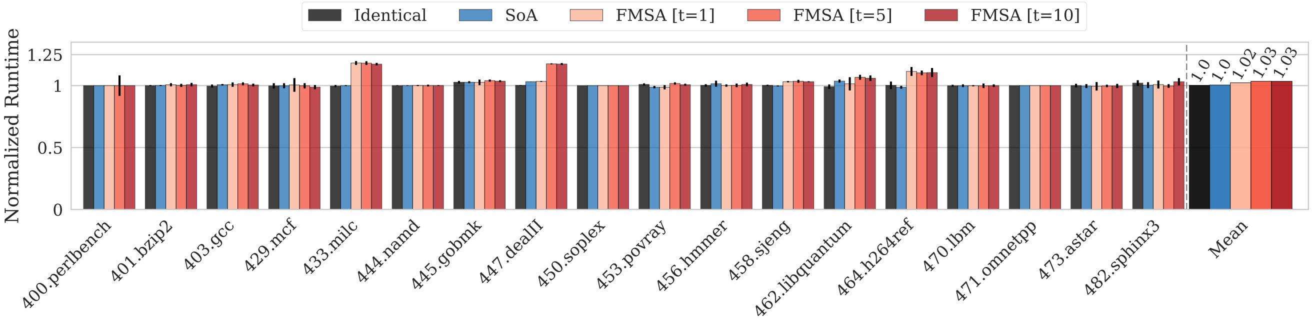

Figure 13. Runtime overhead on the Intel platform. Performance impact is almost always statistically insignificant. For the few benchmarks affected, FMSA merges hot functions.

level implementations are also offered by other production compilers and linkers, such as MSVC [30].

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 [15, 20]. 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, proposed by Edler von Koch et al. [11], exploits structural similarity among functions. 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, they also require that corresponding basic blocks of isomorphic CFGs have exactly the same number of instructions and that corresponding instructions must have equivalent resulting types but may differ in their opcodes or in the number and type of their input operands. If two 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 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.

Although the state-of-the-art technique improves over LLVM's identical function merging, it is still unnecessarily limited. In Section 2, 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.

6.2 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 [21]. Local factoring, also known as local code motion, moves identical instructions from multiple basic blocks to either their common predecessor or successor, whenever valid [2, 18, 21]. Procedural abstraction finds identical code that can be extracted into a separate function, replacing all replicated occurrences with a function call [9, 21].

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 separately 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.

7. 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 3×. 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 22%, with an overall average of about 5.3%, while introducing an average compilation-time overhead of only 20%. 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 can achieve further improvements by integrating the function-merging optimization to a summary-based parallel 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.

References

[1] 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*, 29(22):e3932–n/a, 2017. ISSN 1532-0634. e3932 cpe.3932.

[2] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 159–170, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. .

[3] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.*, 48(5):1073–1082, Oct. 1988. ISSN 0036-1399. .

[4] W.-K. Chen, B. Li, and R. Gupta. Code compaction of matching single-entry multiple-exit regions. In R. Cousot, editor, *Static Analysis*, pages 401–417, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[5] J. Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, pages 20–24, New York, NY, USA, 1970. ACM.

[6] 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*, LCTES '99, pages 1–9, New York, NY, USA, 1999. ACM. ISBN 1-58113-136-4. .

[7] C. de Dinechin. C++ exception handling. *IEEE Concurrency*, 8(4):72–79, Oct. 2000. ISSN 1092-3063. .

[8] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, Mar. 2000. .

[9] 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)*, pages 259–270, March 2007. .

[10] 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*, CGO '10, pages 180–189, New York, NY, USA, 2010. ACM. .

[11] 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*, LCTES '14, pages 85–94, New York, NY, USA, 2014. ACM. .

[12] 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*, PLDI '97, pages 358–365, New York, NY, USA, 1997. ACM. .

[13] 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*, RECOMB'11, pages 85–103, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] D. G. Higgins and P. M. Sharp. Fast and sensitive multiple sequence alignments on a microcomputer. *Bioinformatics*, 5 (2):151–153, 1989.

[15] T. L. C. Infrastructure. MergeFunctions pass, how it works. http://llvm.org/docs/MergeFunctions.html, 2018.

[16] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[17] S. L. Keoh, S. S. Kumar, and H. Tschofenig. Securing the internet of things: A standardization perspective. *IEEE Internet of Things Journal*, 1(3):265–275, June 2014.

[18] 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*, PLDI '94, pages 147–158, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. .

[19] 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*, pages 1–4, April 2012. .

[20] M. Liška. Optimizing large applications. *arXiv preprint arXiv:1403.6997*, 2014.

[21] G. Lóki, Á. Kiss, J. Jász, and Á. Beszédes. Code factoring in GCC. In *Proceedings of the 2004 GCC Developers' Summit*, pages 79–84, 2004.

[22] H. Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[23] 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*, 48 (3):443 – 453, 1970. ISSN 0022-2836. .

[24] P. Plaza, E. Sancristobal, G. Carro, M. Castro, and E. Ruiz. Wireless development boards to connect the world. In M. E. Auer and D. G. Zutin, editors, *Online Engineering & Internet of Things*, pages 19–27, Cham, 2018. Springer International Publishing.

[25] 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*, CGO 2018, pages 163–174, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5617-6. .

[26] U. P. Schultz, K. Burgaard, 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*, LCTES '03, pages 42–50, New York, NY, USA, 2003. ACM. ISBN 1-58113-647-1.

[27] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder. Management of resource constrained devices in the internet of things. *IEEE Communications Magazine*, 50(12):144–149, December 2012.

[28] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981. ISSN 0022-2836. .

[29] SPEC. Standard Performance Evaluation Corp Benchmarks. http://www.spec.org, 2014.

[30] M. V. Studio. Identical COMDAT folding. https://msdn.microsoft.com/en-us/library/bxwfs976.aspx, 2018.

[31] 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.

[32] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson. Using peephole optimization on intermediate code. *ACM Trans. Program. Lang. Syst.*, 4(1):21–36, Jan. 1982. .

[33] 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*, volume 3, pages 161–166 Vol.3, Feb 2004.

[34] L. WANG and T. JIANG. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4): 337–348, 1994. .

[35] V. M. Weaver and S. A. McKee. Code density concerns for new architectures. In *2009 IEEE International Conference on Computer Design*, pages 459–464, Oct 2009.