RollBin: Reducing Code-Size via Loop Rerolling at Binary Level

Tianao Ge
Sun Yat-Sen University
China
getao3@mail2.sysu.edu.cn

Zewei Mo
Sun Yat-Sen University
China
mozw5@mail2.sysu.edu.cn

Kan Wu
Sun Yat-Sen University
China
wukan3@mail2.sysu.edu.cn

Xianwei Zhang
Sun Yat-Sen University
China
zhangxw79@mail.sysu.edu.cn

Yutong Lu
Sun Yat-Sen University
China
luyutong@mail.sysu.edu.cn

Abstract

Code size is an increasing concern on resource constrained systems, ranging from embedded devices to cloud servers. To address the issue, lowering memory occupancy has become a priority in developing and deploying applications, and accordingly compiler-based optimizations have been proposed to reduce program footprint. However, prior arts are generally dealing with source codes or intermediate representations, and thus are very limited in scope in real scenarios where only binary files are commonly provided. To fill the gap, this paper presents a novel code-size optimization `RollBin` to reroll loops at binary level. `RollBin` first locates the unrolled loops in binary files, and then probes to decide the unrolling factor by identifying regular memory address patterns. To reconstruct the iterations, we propose a customized data dependency analysis that tackles the challenges brought by shuffled instructions and loop-carry dependencies. Next, the recognized iterations are rolled up through instruction removal and update, which are generally reverting the normal unrolling procedure. The evaluations on standard SPEC2006/2017 and MiBench demonstrate that `RollBin` effectively shrinks code size by 1.7% and 2.2% on average (up to 7.8%), which respectively outperforms the state-of-the-arts by 31% and 38%. In addition, the use cases of representative realistic applications manifest that `RollBin` can be applicable in practices.

CCS Concepts: • **Software and its engineering → Compilers**.

LCTES '22, June 14, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9266-2/22/06…$15.00
https://doi.org/10.1145/3519941.3535072

Keywords: Code-Size Reduction, Loop Rerolling, Binary Optimization

ACM Reference Format:
Tianao Ge, Zewei Mo, Kan Wu, Xianwei Zhang, and Yutong Lu. 2022. RollBin: Reducing Code-Size via Loop Rerolling at Binary Level. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3519941.3535072

1 Introduction

In the past decades, computer programs have been continuously gaining new features and growing in size and complexity, which together drive the non-stop need for higher computing horsepower and larger memory capacity [2, 14]. As such, for smoothly executing programs and efficiently utilizing the precious resources, especially the memory space and bandwidth, reducing program footprint becomes essential on all computing platforms spanning from servers to embedded systems. For embedded and Internet-of-Things (IoT) devices, code volume is an overwhelming concern, as it directly impacts the chip area and cost, and further influences the overall performance and power [29, 42]. On larger machines, such as desktops, servers and supercomputers, whereas memory capacity is typically much less limited, code size is nonetheless critical for instruction cache (I-cache) performance [43]. Recently, there has been an increasing trend toward unifying libraries, tools, and frameworks to support cross-architecture executions [6, 20], including servers and edge devices, which thus further emphasizes the compacted code across platforms. TensorFlow Lite [40] and BLASFEO [13] are such representative examples actively expanding the machine learning and high-performance computing territories from powerful servers to constrained devices.

Classical techniques, including variable-length instruction encoding [16, 30], code compression [25, 44], and ISA modification [45], are designed to reduce the size of code. Program footprint can also be lessened by compiler-based similar code merging [34] and dead-code eliminating [21, 26].

However, compared to performance boost, code compaction has not been paid much attention until recently [7–9]. Optimal inlining [41] and loop rolling [31] deflate program sizes through source-code optimizations. Nonetheless, these compiler-based techniques work on source code or intermediate representation (IR) and can be exceedingly limited in the following scenarios: 1) whereas with size-related options being provided in modern compilers (such as `-Os` in Clang/LLVM), code size issue is more likely to bother when binaries get deployed, particularly when binaries weren't optimized for code size; 2) source codes are not provided for proprietary reasons, which is a paradigm in commercial situations (e.g., mobile applications submitted on the Google and Apple store); 3) the code is hard to recompile, particularly for legacy applications which have no source available or require a lot of effort to be ported. In addition, for embedded systems, the development may largely rely on assembly programming, thus causing the absence of source code.

Aiming to reduce code size at binary level, we propose `RollBin` to revert the loop unrolling, which is heavily used to optimize execution speed at the sacrifice of program footprint. `RollBin` accepts a binary file as input. First, it analyzes the assembly code of the input file, identifies unrolled loops and then infers the unrolling factor by examining regular memory address patterns. Next, it locates specific instructions which are used as anchor-points for each iteration, and then groups instructions which are supposed to be in the same iteration. Using a customized data dependency analysis, we increase the optimization opportunities by overcoming the limitations of loops with shuffled instructions and loop-carry dependencies. Finally, it rolls these loops into the ones with smaller code size and produces a new shrunk binary file. In addition, profiling data is introduced to guide the loop rerolling strategy, maintaining the performance of the optimized binary.

The contributions of this paper are:

- we highlight the critical need of performing binary-level optimizations to reduce code size and propose a novel design `RollBin` to systematically identify and reroll the expanded iterations.
- through fine-grained iteration probing and optimized instruction clustering, our design efficiently covers a larger portion of unrolled loops than prior arts, bringing in more rerolling opportunities.
- the evaluations on benchmark suites and real applications demonstrate that our design reduces code size effectively without source code, outperforming the state-of-the-arts.

The paper is organized as follows. Section 2 introduces the background and motivates the binary-level code compaction. Section 3 elaborates the proposed design. Section 4 presents the experimental methodology, and Section 5 analyzes our experimental results. Related work is covered in Section 6. The paper is concluded in Section 7.

2 Background & Motivation

To compile a program, the compiler's front-end translates the source code into architecture-independent IR. Next, the middle-end, e.g., `opt` of LLVM, performs multiple optimization passes to augment the IR. Then, the compilation process is concluded by the back-end to translate the IR to binary machine code, which is finally distributed and executed on the target platform. As a classical optimization technique, loop unrolling is frequently used in compilation and works directly on IR, but it greatly affects the final binary code in both performance and code size.

In the remainder of this section, we first introduce the loop unrolling optimizations in the compilation. Then, we motivate the binary-level loop rerolling design.

2.1 Loop Unrolling

Loops are the predominated structures in almost all programs, and thus commonly transformed by compilers for performance improvement. Unrolling is a representative transformation to accelerate the loop execution. With unrolling, we replicate the original loop body multiple times, and then correspondingly adjust the terminating conditions and iterating step, thus amortizing the branching overhead. The number of replication times is called the *unrolling factor*, and the expanded loop is termed as *unrolled loop* (conversely, the original loop is often termed *rerolled loop*). Apparently, unrolling increases the number of instructions, thus potentially enabling further optimizations such as instruction reordering, which intermingle the instructions between loop iterations.

To illustrate the unrolling, Figure 1 presents an example loop, which is extracted from SPEC2006 and simply performs a cumulative sum of multiply-and-subtract calculations. With an unrolling factor of two, the loop in Figure 1(a) is repeated twice to transform into the assembly in Figure 1(b). Among the assembly sequences, instructions L1 - L10 represent the loop body, which consists of two iterations; the final ones L11 - L13 form a *loop latch* to direct the loop to repeat or terminate. Clearly, the latch part involves the actions of update (L11) and comparison (L12), which are actually linked by an *induction register* (`%rcx`). In addition to advancing the loop iterations, the *induction register* is exploited for data movement in the loop body, e.g., `%rcx` is used by the `movss` instructions (L1 - L2). Specifically, `%rcx` is used by `movss` at L1 for retrieving data at address (`%rdx+4*%rcx`) to the register `%xmm2`, which then disseminates through register usage (L3). The dissemination shapes a data dependency chain, as depicted by Figure 1(c), which reflects data flow traces within the loop body.

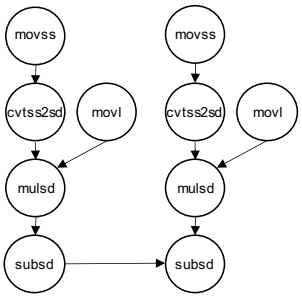
```
for (int row = m() - 1; row >= 0; --row) {
    somenumber s = b(row);
    for (unsigned int j = cols->rowstart[row];
         j < cols->rowstart[row + 1]; ++j) {
      s -= val[j] * v(cols->colnums[j]);
    }
    v(row) += s * om / val[cols->rowstart[row]];
}
```

(a) Original code with a nested loop.

```
.Ltmp32364:
1   movss     (%rdx,%rcx,4),    %xmm2
2   movss     0x4(%rdx,%rcx,4),%xmm3
3   cvtss2sd  %xmm2,            %xmm2
4   movl      (%rax,%rcx,4),    %esi
5   mulsd     (%rbx,%rsi,8),    %xmm2
6   movl      0x4(%rax,%rcx,4),%esi
7   cvtss2sd  %xmm3,            %xmm3
8   mulsd     (%rbx,%rsi,8),    %xmm3
9   subsd     %xmm2,            %xmm1
10  subsd     %xmm3,            %xmm1
11  addq      $0x2,             %rcx
12  cmpq      %rcx,             %rbp
13  jne       .Ltmp32364
```

loop body: 1–10; loop latch: 11–13

(b) The sequence of assembly code with two unrolled iterations in the loop body.



(c) Data dependency graph of instructions in the loop body.

Figure 1. A loop example extracted from `482.sphinx3` in SPEC2006.

2.2 Reroll to Reduce Code Size

Essentially, loop unrolling attempts to enhance a program's execution at the expense of its code volume, which thus may exacerbate the issue of code size on resource-constrained systems. From Figure 1(b), we can see that unrolling introduces five extra instructions (L2, L6-8, L10), i.e., a 62.5% increase (8 instructions to 13). We constantly observe this phenomenon on standardized benchmarks including SPEC and MiBench, which are exhibiting code expansion by 3.2% - 36.7%. In view of this, it is critical to solving the size issue stemming from unrolling. Apparently, if we revert the unrolling (i.e., rerolling), then the code size can be effectively decreased. For the example in Figure 1, rerolling achieves a reduction of 38.5% (i.e., 13 instructions to 8). One straightforward implementation is to regenerate the code by compiling with size-targeted options like `-Os` to avoid or curtail unrolling, which is nonetheless impractical in many scenarios where only binary formats are being accessible.

To the contrary, binary-level rerolling requires no source code or IR, thus significantly extending the applicability. In principle, the workflow of rerolling at binary level should be: 1) recognize loop and its unrolling factor; 2) identify the iterations; 3) fold the unrolled loop. However, the steps can be extremely challenging in implementation:

— *C1*: at binary level, the information stored in instructions is fragmented and hidden. As illustrated in Figure 1(b), the expanded loop is dissimilar to the source code or IR, which possesses high-level semantics like loop modules or represents in static single assignment (SSA) format. Accordingly, binary-level rerolling is obliged to systematically process the raw instructions to retrieve critical clues, e.g., inductions and unrolling factors.

— *C2*: instruction patterns are insufficient to identify the iterations. In unrolling, multiple iterations are presented with instructions being mixed up, which thus disrupts the usual instruction-based analysis. Furthermore, the expanded instructions are commonly rescheduled as a routine follow-up of unrolling. For instance, Figure 1(b) shows the interleaved and reordered instructions from two iterations. Therefore, instead of relying on instructions, binary unrolling necessitates more fine-grained investigations, which can be based on register usages and address sequence.

— *C3*: even with fine-grained analysis, iteration identification can be burdensome, considering the across-iteration data dependency. As depicted by Figure 1(c), the two iterations are correlated, hindering the automatic procedure of iteration restoring. Apparently, a method should be proposed to break up the correlation to progress restoring.

Although with the above challenges, binary-level rerolling attracted attentions recently to identify and restore the iterations. `SubDDG` [18] and `SuffixTree` [35] are such representative designs, which address the challenge *C1*. Particularly, `SubDDG` resorts to iteration-level data dependency graph to check whether a loop can be rerolled, and `SuffixTree` identifies each iteration by finding consecutively repeated sequences of instructions. Clearly, the two designs are ineffective to handle challenges *C2* and *C3*, thus missing plenty of rerolling opportunities. Figure 2 reports the rerolling results of both the illustrated snippet in Figure 1 and the whole program of 482.sphinx3. It indicates that these challenges limit the effect of existing binary-level rerolling techniques.

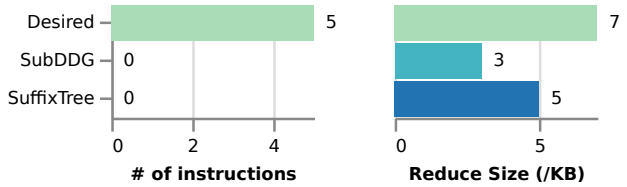


Figure 2. The reduction in instruction count and code size for the motivating example and `482.sphinx3` using existing techniques and the comparison to desire.

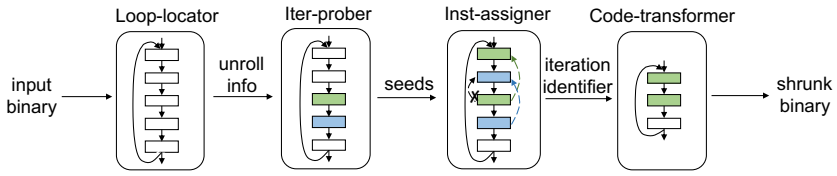


Figure 3. Overview of RollBin.

3 Binary-level Loop Rerolling

In this section, we present RollBin, a novel code reduction technique that works at binary level to effectively capture and fold up unrolled loops. To achieve this, RollBin scrutinizes the binary program to locate the loop snippet, analyzes the registers and addresses to unveil the iterations, and eventually rolls back the loops to compact code. Whereas focusing on code size, RollBin extends the design to support profiling guidance and binary layout rearrangement, which respectively helps preserve desired performance and shrink the final executable.

3.1 Overview

RollBin takes binary files as input, and produces a new executable with reduced code size after conducting loop-based optimizations. Around the loop identification and transformation, the whole procedure can be divided into four phases, each of which corresponds to one modular component of RollBin . As illustrated in Figure 3, RollBin fulfills the desired reduction by going through multiple stages: Loop-locator to identify the loops, Iter-prober to anchor the iterations, Inst-assigner to assign each instruction to its iteration, and Code-transformer to revise the code.

As the forefront phase, Loop-locator aims to recognize the unrolled loops via the induction register, which marks one potential loop as introduced in Section 2.1, together with its associated memory operands, and further obtains the critical unrolling factor to decide the iteration times. With the induction and factor being carried onward, Iter-prober examines the instructions to filter out those depending on the induction. The selected instructions, termed as seeds, are essentially signaling the origins of iterations in the unrolled loops. Thus they are forwarded to next phase Inst-assigner to pinpoint and cluster the instruction sequence of each iteration. The locating is realized by investigating data dependency propagating through seed instructions. After clustering, the final stage Code-transformer comes into play to fold up the iterations to counteract the native unrolling.

3.2 Identify Unrolled Loops

In this phase, the Loop-locator component of RollBin intends to prepare for investigating the loops in the upcoming stages, and thus emphasizes on delimitating the loop regions and conjecturing the vital attributes. Whereas being

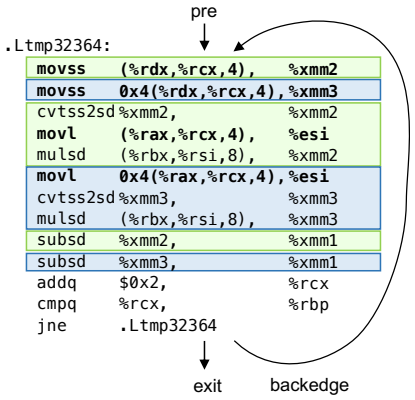


Figure 4. The sequence of assembly code of the motivating example. The instructions belong to different loop iteration are highlight in different colors.

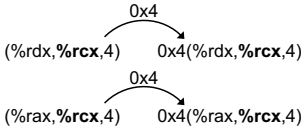


Figure 5. Two monotonically increasing sequences extracted from the motivating example. They are based on the same induction rcx and grow by 0x4.

characterized by repeated sequences, unrolled loops distinguish from usual replicated lines on the regular memory patterns and final wrapping-up instructions, which are thus exploited for loop identification. Therefore, RollBin identifies unrolled loops by detecting regular memory address sequences to eliminate interference from instruction order.

In detail, RollBin first detects the jump or branch instructions to locate the loops, and then selects the inductions, which are carried out along the loop execution. Specifically, RollBin focuses on the inner-most loops within a single basic block because compilers tend to unroll small loops with simple control flow. Next, the inductions[1] are utilized to backtrack the associated memory addressing within the loop. Those addresses form a regular sequence, and meanwhile the associated instructions are only differing on the

[1]It should be noted that whereas multiple inductions may exist in one single loop, any one is sufficient for the backtracking.

```
.Ltmp1                                  .Ltmp1
  movl   $0x0,  (%rdx)                    movl   $0x0,  (%rdx)
  movl   $0x0,  0x4(%rdx)                 movl   $0x0,  0x4(%rdx)
  addq   $0x20, %rdx                      movl   $0x0,  0x8(%rdx)
  movl   $0x0,  -0x18(%rdx)               movl   $0x0,  0xc(%rdx)
  movl   $0x0,  -0x14(%rdx)               movl   $0x0,  0x10(%rdx)
  movl   $0x0,  -0x10(%rdx)               movl   $0x0,  0x14(%rdx)
  movl   $0x0,  -0xc(%rdx)                movl   $0x0,  0x18(%rdx)
  movl   $0x0,  -0x8(%rdx)                movl   $0x0,  0x1c(%rdx)
  movl   $0x0,  -0x4(%rdx)                addq   $0x20, %rdx
  cmpq   %rdx,  %rcx                      cmpq   %rdx,  %rcx
  jne.   .Ltmp1                           jne.   .Ltmp1
```

Figure 6. Example of rearranging shuffled instructions. The rearrangement moves the instruction operating on the induction to the leading of the loop latch, yielding a monotonically increasing sequence.

displacement of the address, but completely identical on opcode, index register, base register and scaling factor. In the example from Figure 4, the memory addresses in the first two instructions `movss` form a regular sequence with an interval of 4, so do another two instructions `movl`. Figure 5 lists the extracted memory accesses based on the induction `rcx`. Apparently, the sequence is monotonically increasing with a fixed interval `0x4`, manifesting a clear unrolling pattern. Whereas being desired, monotonicity is not always there because of the instruction rescheduling performed in compilations. As a result, an extra step is necessitated to rearrange the accesses to expose a monotonous sequence. While the counter-updating instruction can be inserted in the loop body, thus breaking the monotonous sequence, `RollBin` relocates the instruction to the leading of the loop latch and then adjusts the memory access offset. To provide a clearer explanation, Figure 6 shows another example with instruction rescheduling. The instruction `addq`, performing integer addition on the induction `rdx`, makes the base addresses of instructions different, preventing them from establishing a monotonically increasing sequence. `RollBin` moves the instruction `addq` and modifies correlative memory access to discover the monotonicity.

In addition to checking unrolled loops, the collected addresses can be further used to infer the unrolling factor, which directly indicates the iteration times. For the example in Figure 4, there are two addresses in a monotonous sequence which are `0x4` apart, conveying a two-time unrolling. Actually, the address-based factor needs to be aligned with the increment/decrement interval, which co-exists with the induction as an immediate (e.g., "`0x2`" in the instruction "`addq $0x2 %rcx`"). Generally, the immediate must be divisible by the unroll factor; otherwise, the identified loop is disregarded as unrolled, terminating the rerolling procedure.

3.3 Anchor the Iterations

Once deciding the probable unroll factor, we next traverse all memory-accessing instructions related to any induction,

```
movss  (%rdx,%rcx,4),    %xmm2   #(0-0)/4=0 → iter 0
movss  0x4(%rdx,%rcx,4), %xmm3   #(4-0)/4=1 → iter 1

movl   (%rax,%rcx,4),    %esi    #(0-0)/4=0 → iter 0
movl   0x4(%rax,%rcx,4), %esi    #(4-0)/4=1 → iter 1
```

Figure 7. Example of two groups of seed instructions which are assigned with a numeric identifier.

which are further analyzed to speculate their corresponding iterations they belong to. The memory addresses are requested to form a monotonous sequence with a fixed interval and to be the same length as the probable unroll factor. So the instructions associated with a memory-access sequence are equivalent across iterations and each one is assigned to the appropriate iteration. The iteration residence is denoted using a numeric identifier, which is calculated as $(displacement - start)/step$. More specifically, the displacement is the integer added to the memory address while the *start* and *step* correspond to the first displacement and the interval in the memory sequence, respectively.

Figure 7 shows how we anchor the seed instructions in the motivating example. There are two monotonically increasing sequences with a length of two. Two instructions in each sequence are assigned to iteration 0 and iteration 1, respectively, because the loop is unrolled into two iterations. These marked instructions, also known as seed instructions, essentially communicate the original information of iterations in the unrolled loops. The corresponding iterations of other instructions which have no memory access are retrieved using these seeds, as we discuss in the following subsection.

3.4 Cluster the Instructions

With the collected loop attributes, we then strive to reconstruct the iterations by putting back each instruction to its native loop instance. The reconstruction is accomplished using customized data dependency analysis as follow.

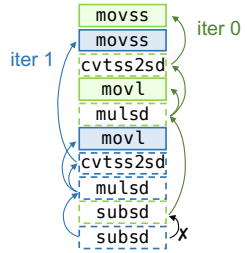


Figure 8. Data dependency analysis propagates the iteration number and intercepts by wall instructions. Seed instructions are represented by solid-line boxes.

Naturally, data dependency is employed to cluster the instructions into their native loop iterations. To this end, each instruction is expected to be assigned an iteration number,

Algorithm 1: Cluster all instructions into loop iterations.

```
Function ClusterInstructions(LoopInsts):
    foreach inst ∈ LoopInsts do
        if IsSeedInst(inst) then
            UpdateIteration(inst);
        else if CheckDependency(inst) then
            i ← GetLargerIteration(inst);
            AssignIteration(inst,i);
            UpdateIteration(inst);
Function UpdateIteration(inst):
    i ← GetIteration(inst);
    foreach dependInst ∈ GetDependInsts(inst) do
        if IsWallInst(dependInst) then
            continue;
        h ← HasIteration(dependInst);
        if not h then
            j ← GetIteration(dependInst)
        if h or i > j then
            AssignIteration(dependInst,i);
            UpdateIteration(dependInst);
```

which was already attached to the seed instructions of each iteration. The data dependency graph is constructed based on the register usage in context. The instructions with register reading rely on their definition instruction until the value in the register is updated. Along the data dependency graph, the iteration number propagates among instructions to progress the clustering. Figure 8 presents the data dependencies in the motivating example. The third instruction `cvtss2sd` depends on the first instruction `movss` because the operands used in `cvtss2sd` are defined in `movss` most recently, which enables the iteration number of `movss` to be propagated to the `cvtss2sd`. However, the clustering in practices is more complicated, because of the fact that data dependencies usually span multiple iterations. One representative scenario is that a variable is shared among iterations, and this causes results calculated in the preceding iteration to be used in the next round. For instance, the last two `subsd` instructions in Figure 8 operate the same register `xmm1` to take a cumulative subtraction. Even though the latter one depends on the former one, they should not be grouped into the same iteration.

To tackle the across-iteration dependency, we introduce special wall instructions, which intercept the undesired number propagation to break up the dependency. Particularly, wall instructions are defined as either seeds or those recursively depending on other wall instructions. Algorithm 1 outlines the iteration number propagation strategy taking advantage of wall instructions. In detail, the process works by scanning all instructions in the loop and then performs a bottom-up propagation to cluster instructions. Instructions are assigned with iteration numbers via their dependencies. When an instruction reuses the calculation results in previous iteration, it will depend on different iterations, then the larger iteration number will be chosen to propagate. Unless the instruction is a wall instruction, all dependencies will be checked and updated recursively once it is assigned with an iteration number.

3.5 Transform the Code

After completing the loop reconstruction, `RollBin` proceeds to eradicate all or partial iterations to reroll the loop. However, before activating the rerolling, `RollBin` should validate the reconstructed iterations to ensure they are isomorphic. Iterations are deemed as isomorphic only if they have identical structure and each instruction is of the same opcode and operands with its counterpart. Nonetheless, the isomorphic criteria might be exceedingly rigorous, considering that iterations can be semantically equivalent but structurally different, which is typically caused by compiler-directed code adjustments. As we focus on solving the dependency issue when grouping, we adopt simple transforming in our design by removing additional `mov` instructions which are redundant. Then we ensure that any two groups are isomorphic by the following rules: 1) the number of instructions in each group is the same; 2) the corresponding instructions in each group have an identical opcode; 3) they contain only jump instructions or operations on inductions in the loop latch.

Once passing the validation, `RollBin` advances to reroll the loops, which involves iteration removal and latch update, which are pretty straightforward to implement. However, emphasizing solely on code size inevitably poses a risk of significant performance degradation, hindering the wider adoption of `RollBin` . To overcome the issues, `RollBin` further incorporates performance profiling data to guide the loop rerolls. In principle, rerolling degree should be negatively proportional to the loop's criticality on performance, e.g., light-or-none rerolling on loops predominating the overall execution. Specifically, `RollBin` is extended with a threshold `RerollFraction` to control loop rerolling with profiling. Based on the ascending order of execution frequency, the top `RerollFraction` of loops won't be rerolled, while the other loops with lower executed frequency will be rerolled completely. Once getting folded, the loop instructions will be generated into a relocatable file, whose `.text` section shrinks in size.

3.6 Implementation

Figure 9 shows the pipeline of our technique. It bases on the LLVM Compiler Infrastructure [22] and BOLT binary optimizer [27], which help handle disassembly and modification of binary files. The Linux Perf tool is used to measure the number of executed times for loops. The profiling data obtained via sampling will be processed and mapped to assembly code. To reduce the size of text section during binary

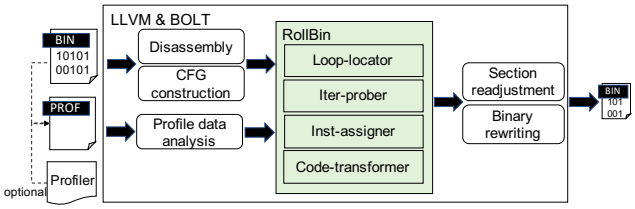


Figure 9. The processing pipeline of our implementation. The inputs include the target binary and optional profiled data. The output is a program with smaller code size.

rewriting, relocation information is required to adjust function position. Relocation can be remained when linked with specific options, such as `-emit-relocs`. For stripped binaries with no text relocation, reconstruction of relocation information [28] can be applied to enable function position modification.

Simply replacing the text section tends to leave space bubbles, wasting storage. To fix the issue, we extend `RollBin` to adjust the layout after rerolling loops for storage reduction in the final executable. We reimplement the binary rewriting module in BOLT and try to adjust all section offsets to a compact structure to discharge these empty spaces. Corresponding information such as program header table and section table is updated to ensure the program can be loaded in memory and find the binary entry correctly.

4 Evaluation Methodology

4.1 Experimental Setup

Hardware & Software: We conduct experiments on Linux-based servers featuring AMD EPYC 7742 CPU, which is of 2.25 GHz (3.4 GHz boost) frequency, private L1/L2 of 94 KB/512 KB, and shared L3/memory of 256 MB/256 GB. For the software, OS is CentOS 7.9 (kernel version 3.10.0), supporting Linux's perf (version 4.15.18), and compiler is Clang/LLVM 13.0.0.

Benchmarks: To evaluate `RollBin` in a variety of scenarios, we run experiments on different benchmark suites, including SPEC2006 [17], SPEC2017 [5], and MiBench [15], which are the most widely used workloads in general-purpose and embedded systems. In addition, TSVC [1], a loop-heavy micro-benchmark, is included for better understanding the benefits gained by `RollBin` rerolling. TSVC consists of 151 kernels, each containing a single loop. Moreover, we perform use-case studies on TensorFlow Lite [40] and BLASFEO [13], which are two realistic applications of machine learning and linear algebra, respectively. Table 1 briefly summarizes the benchmarks.

In terms of binary-code construction, we build the workloads using Clang/LLVM with appropriate optimizations. For SPEC 2006/2017, MiBench and TSVC, we repeatedly compile by enumerating `-Os'` (`-Os` with loop unrolling being enabled), `-O2` and `-O3`, which may greatly affect the program footprint. To build TensorFlow Lite and BLASFEO, we instead adopt the officially recommended options to conform to the real usages.

Table 1. Evaluated Applications.

Category	Suite	Domain
Standard	SPEC2006	General-Purpose
	SPEC2017	General-Purpose
	MiBench	Embedded System
Micro-bench	TSVC	Vectorization
Real-app	TensorFlow Lite	Machine Learning
	BLASFEO	Linear Algebra

4.2 Designs and Metrics

Contending designs: To evaluate the effectiveness of our proposed design, we compare `RollBin` against the baseline and prior arts, which cover both binary-level and IR-level code size reduction techniques. Primarily, we study and compare the following approaches:

- `Baseline`. Default Clang/LLVM with selected option, which can be `-Os'`, `-O2` or `-O3`.
- `RollBin`. The proposed design, which applies binary-level rerolling atop of the `Baseline` by analyzing memory addresses and data dependencies.
- `SubDDG` [18]. A binary-level rerolling technique which operates on independent loops
- `SuffixTree` [35]. A binary-level rerolling technique using suffix trees to identify repeated instructions.
- `RoLAG` [31]. The state-of-the-art rolling approach at IR-level.

Metrics: As the predominant metric to measure rerolling effect, code size is denoted by the text segment of the binary file. To quantify code reduction, we first get the absolute value in KB, by calculating the difference between `Baseline` and the studied design, and then get the percentage as:

$$Reduction\ (\%) = \frac{Size_{diff}}{Size_{base}}$$

To evaluate performance, we generally use the profiled execution time, and then convert into a slowdown ratio. The execution time is averaged across ten repeated runs. As for the final executable, we directly consider the raw file size.

5 Results and Analysis

Following the aforementioned methodology, we conduct experiments to study the listed design approaches over the pertinent workloads. This section presents and analyzes the collected results to demonstrate `RollBin`'s effects from multiple facets, including code size, executable size, and performance.

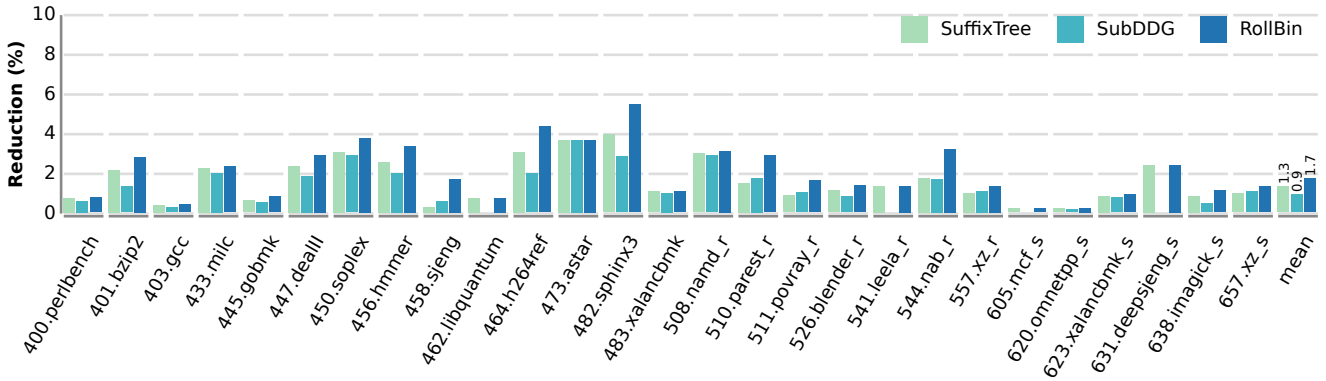


Figure 10. Code size reduction on SPEC2006/2017 over `-Os'`. Only a subset of the workload is shown. Out of 43 benchmarks: 40 shrink in size, 3 remain unchanged. The most significant reduction achieves on `482.sphinx3` (5.4%). The average reduction on all 43 benchmarks is 1.7%, outperforms the state-of-the-art by 31%.

5.1 Code Size

We evaluate the `RollBin` design across SPEC2006, SPEC2017, and Mibench benchmark suites, and compare to the closest state-of-the-arts `SubDDG` and `SuffixTree`, which are likewise rolling loops at binary level. The code reduction results of SPEC and MiBench are respectively reported[2] in Figure 10 and Figure 11 .

From Figure 10, we can see that `RollBin` is effectively reducing code size of SPEC benchmarks, and performs better than `SubDDG` and `SuffixTree` in most cases. Relative to the `Baseline`, `RollBin` achieves an average reduction of 1.7%, beating the 1.3% and 0.9% of the two contending designs. Particularly, `RollBin` reports the maximal reduction of 5.4% on *482.sphinx3*, which is a loop-heavy benchmark favoring our rerolling techniques. Note that on certain benchmarks like *462.libquantum*, *631.deepsjeng_s* and *541.leela_r*, `SubDDG` shows very trivial effect, whereas `RollBin` still reaches considerable reductions. These are the cases that loop-carry-dependency exists among the majority of loops, such as accumulation of arrays and calculation across multiple iterations, causing iterations in an unrolled loop can't be distinguished by data dependency graph. On the other hand, `RollBin` well addresses the issue by introducing the concept of wall instructions (see Section 3.4). Meanwhile, although `SuffixTree` is capable to reroll more loops than `SubDDG`, it only benefits loops with repetitive code structure. Since `SuffixTree` is seriously restricted by the order of instructions, the normal instruction flow is disrupted and further looping opportunities are missed, which can be reflected on *482.sphinx3* and *544.nab_r*. Instead, `RollBin` well tolerates the instruction shuffling, and hence realizes remarkable reductions on these applications (38.4% and 82.4% better than `SuffixTree` on *482.sphinx3* and *544.nab_r*).

[2]Due to the limited space, the figures show a subset of the workloads, together with a mean of the complete set.

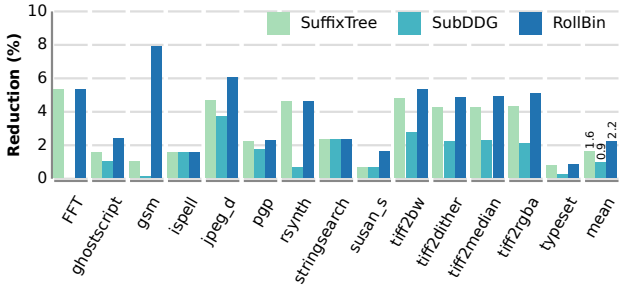


Figure 11. Code size reduction on MiBench over `-Os'`. Only a subset of the workload is shown. Out of 24 benchmarks: 14 shrink in size, 10 remain unchanged. The most significant reduction achieves on `gsm` (7.8%). The average reduction on all 24 benchmarks is 2.2%, outperforms the state-of-the-art by 38%.

Similarly, Figure 11 presents the code size results of MiBench. Not unexpected, `RollBin` attains more promising reductions on those embedded applications, which are extremely stressing on code size. Overall, `RollBin` smoothly reduces the code size of most applications and reports an average reduction of 2.2% (up to 7.83%), which are substantially better than `SubDDG` and `SuffixTree`. It is noteworthy that for partial applications with very limited amount of instructions, none of the three evaluated designs comes into operation. *adpcm_c* is one such representative example having only two .c files with less than 500 instructions. We further repeat our experiments over different optimization levels: `-O2` and `-O3`, and summarize the code size reduction in Table 2. Compared with `-Os'`, the complex transformations in higher optimization levels lead to the dramatic expansion of binary and make it harder to reroll, which results in a lower reduction by loop rerolling. However, it indicates that RollBin outperforms the other two approaches at these optimization levels.

Table 2. The average code size reduction for SPEC2006/2017 and MiBench over different optimization levels.

Suit	Flag	SubDDG	SuffixTree	RollBin
SPEC2006/2017	-O2	0.40%	0.77%	0.93%
	-O3	0.39%	0.74%	0.89%
MiBench	-O2	0.28%	0.35%	0.50%
	-O3	0.28%	0.33%	0.50%

To better understand `RollBin`'s efficacy, we further collect the fundamental statistics of loop rerolling, which are `#Lps` and `Size` as listed in Table 3. `#Lps` and `Size` respectively denote the amount of the successfully rerolled loops and the absolute size of code reduction. From the table, we see that `RollBin` rerolls more loops than the peer designs, therefore bringing in a more significant size reduction. For example, *510.paresr_r* achieves `#Lps` of 1389 and `Size` of 205 KB, which are twice the quotas of `SuffixTree` and `SubDDG`. Overall, `RollBin` is also effective on absolute reductions and respectively reduces total code size by 173 KB, 428 KB and 65 KB across SPEC2006/2017 and MiBench.

Table 3. Detailed statistics of loop rerolling on SPEC2006/2017 and MiBench. `#Lps` is the amount of the successfully rerolled loops, and `Size` represents the absolute size of code reduction in KB.

Suite	Benchmark	SubDDG		SuffixTree		RollBin	
		#Lps	Size	#Lps	Size	#Lps	Size
SPEC2006	400.perlbench	41	4	48	5	51	5
	401.bzip2	8	0	11	1	13	1
	403.gcc	56	4	66	7	76	8
	447.dealII	383	38	454	48	544	60
	...			...			
	450.soplex	49	9	53	9	67	11
	456.hmmer	49	4	68	5	80	7
	464.h264ref	100	9	131	15	166	21
	483.xalancbmk	255	17	261	18	262	19
	sum (total)	1111	100	1332	137	1551	173
SPEC2017	508.namd_r	420	27	429	28	435	29
	510.parest_r	967	123	934	105	1389	205
	511.povray_r	79	7	81	6	105	11
	526.blender_r	590	53	690	75	811	92
	...			...			
	620.omnetpp_s	17	1	20	1	26	1
	623.xalancbmk_s	215	15	223	17	239	18
	638.imagick_s	56	7	78	14	99	18
	657.xz_s	13	1	12	1	14	1
	sum (total)	2758	264	2920	289	3654	428
MiBench	bitcount	0	0	0	0	0	0
	ghostscript	59	7	76	11	108	17
	gsm	1	0	2	0	6	2
	jpeg_d	29	4	39	6	47	7
	...			...			
	pgp	27	2	30	3	32	3
	tiff2bw	31	3	46	6	55	6
	tiff2dither	28	2	43	5	52	6
	typeset	16	0	18	3	25	3
	sum (total)	287	26	403	51	508	65

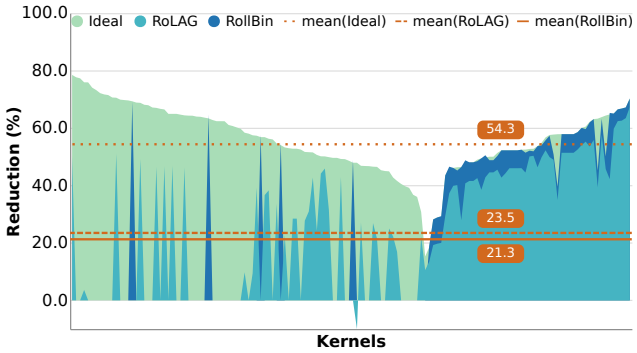


Figure 12. Code size reduction achieved by `RoLAG` and `RollBin` and a comparison with ideal case across TSVC kernels, sorted by `RollBin` and `RoLAG`'s reduction numbers.

5.2 Compare to IR-level Rerolling

Alternatively, loop rerolling can be performed at IR-level, albeit greatly limiting the scope to specific situations with source codes available. To reinforce the evaluation, we next compare `RollBin` against `RoLAG`, which is the most recent IR-based rolling approach. To be consistent on evaluation, we use the same TSVC micro-benchmark to assess the approaches. To align with `RoLAG`, the inner loops are forced to be unrolled by 8 and compiled with `-Os`.

Figure 12 shows the reduction on all TSVC kernels obtained by each technique and the gap from ideal situations where loops are completely rerolled. On average, `RollBin` and `RoLAG` achieved similar reductions while `RoLAG` outperforms `RollBin` by a small margin about 2 percentage point (21.3% for `RollBin` and 23.5% for `RoLAG`). Even without source code level information, `RollBin` still performs well on loop rerolling. For the kernels rerolled by both, `RollBin` attains better results than `RoLAG`. This is because `RollBin` rerolls loops completely, while `RoLAG` partially rerolls loops and creates a new inner loop which takes up extra space.

Besides evaluating the effectiveness, these results also indicate the limitations of both `RollBin` and `RoLAG`. The most prominent of failure are the loops with multiple basic blocks, which both `RollBin` and `RoLAG` fail to handle. Inter-iteration loop optimizations, such as tiling and vectorization, can also disrupt the pattern between iterations, thus making it difficult to extract the original loop-body from the binary. The remaining cases are caused by the isomorphic of iterations. Optimizing certain iterations breaks the isomorphic, and `RollBin` identifies them as non-unrolled loops. Such cases can be covered by adding more code-transforms to `RollBin`, e.g., semantically identical instructions replacing.

5.3 Case Study: Real Applications

In addition to the classical benchmark suites, we extend the evaluation by incorporating two real applications, TensorFlow Lite and BLASFEO, which are widely used in embedded and edge devices. TensorFlow Lite is a lightweight deep

learning framework for mobile devices. Containing a rich of machine learning layers like convolution and pooling, TensorFlow Lite is predominated by loop calculations. In particular, we select a C++ pre-built native binary provided by the official which is used to benchmark a TFLite model and its individual operators. Evaluation is performed on the linux x86-64 version binary with default flags (i.e., `-O2`) [39], and the results show that `RollBin` deflates the code by 81 KB or by 1.9% over 2024 unrolled loops.

The other application, BLASFEO, is a library of BLAS- and LAPACK-like routines optimized for embedded infrastructures. At its core, BLASFEO employs plenty of hand-crafted assembly-coded dense linear algebra kernels which are challenging for end users to perform further revisions. Directly fed into the assembly, `RollBin` can continue optimizing to shrink code size. To faithfully model the real usage, we build the library with the default option (i.e., `-O2 -mavx2 -mfma`) [12]. With the rerolling technique of `RollBin` , BLASFEO successfully folds up 669 loops out of 3430, compacting code size by 24 KB (i.e., a 1.6% reduction). It demonstrates that `RollBin` can be applied to shrink real applications.

5.4 Executable Size

Despite highlighting instruction code, `RollBin` is expected to influence the eventual executable, and additionally enables to adjust the post-rerolling sections of the application. For evaluation, we measure the final executable file size among the benchmark suites, and list the overall results in Table 4. Whereas possibly compromised by the OS-layer routines (e.g., 4K alignment for segment loading), `RollBin` still shows an observable file-size reduction. It results in average reductions of 0.9% (total 647 KB) on SPEC2006/2017 and 1.1% (total 72 KB) on MiBench. For the remaining benchmarks who contain one binary file, the largest reduction is around 84 KB, achieved on TSVC.

Table 4. Binary size reduction achieved by `RollBin` on all evaluated applications. Abbreviations used: MEAN and MAX, mean and maximum file size reduction; PMEAN and PMAX, mean and maximum percentage file size reduction; SUM, total file size reduction.

	MEAN	PMEAN	MAX	PMAX	SUM
SPEC2006/2017	17 KB	0.9%	218 KB	3.3%	647 KB
MiBench	3 KB	1.1%	18 KB	4.2%	72 KB
TSVC	13 KB	10.0%	N/A	N/A	N/A
TensorFlow Lite	84 KB	1.5%	N/A	N/A	N/A
BLASFEO	25 KB	1.4%	N/A	N/A	N/A

5.5 Performance

Even though endeavoring to condense program code, loop rerolling may adversely affect performance. Consequently, we further gauge `RollBin`'s intertwining effects on code size and execution performance, by tuning the profiling-based knob to configure the trade-offs. We adjust the loop folding degree, denoted by `RerollFraction` threshold to direct `RollBin` to only operate on the non-performance-critical loops, equaling to protect the critical ones. Figure 13 conveys that performance-unaware strategy (i.e., set threshold to 1) downgrades performance by 15% on TSVC. With the help of profiling guidance, `RollBin` succeeds to lower the slowdown to a very moderate level (~1%) when threshold is set to 0.8, meanwhile maintaining comparable size reduction of 12%. On 456.hmmer (SPEC2006) and TensorFlow Lite, due to most rerolled loops with a single block not being hotspots, varying thresholds have little impact on performance. Apparently, trade-off optimizations can be explored to exert `RollBin`'s full potential, but this is beyond the focus of this paper and thus left for future work.

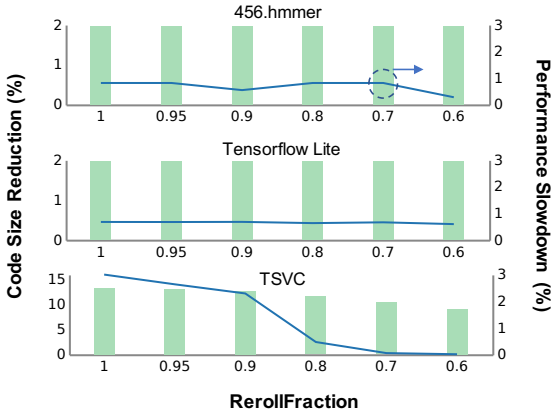


Figure 13. Code size reduction and performance slowdown under varying `RerollFraction` thresholds.

6 Related Work

Code size optimization: Optimizations for code size have been a basic part since the birth of the compiler, but without much attention. Previous approaches reduce code size by removing the redundant or dead codes segment, such as common subexpression elimination [3, 24] and deleting unnecessary code [4, 21]. Function merge [23, 32, 33, 43] is used to identify and merge similar subsequences in different functions. F3M [36] is the state-of-art function merge technique which uses a hash-based fingerprint to summarize functions. Function inlining [9, 19] and loop unrolling [10] improve performance at the expenses of increasing code size, which are essential in code size optimizations. Theodoridis et al. [41] introduced a novel inlining search space formulation which allows massive space reductions to exhaustively find the optimal inlining decisions. RoLAG [31] creates loops out of straight-line code based on a bottom-up graph alignment solution. These code size optimizations all work on IR and

necessitate the compilation of source code. For the programs written in assembly, pre-built binaries and library files, they suffer from the lack of source information and are unable to be optimized. `RollBin` performs at binary level and is orthogonal to these works.

Binary optimization: In recent years, binary optimizers have become popular, such as Propeller [37], Janus [46] and Halo [11]. These tools use dynamic runtime information to reduce the overhead on cache and branch prediction by adjusting the code layout in the file. Bolt [27] is an open-source post-link optimizer built on top of the LLVM framework. It can boost the performance of real-world applications with both profile-guided optimizations (PGO) and link-time optimizations (LTO). However, these optimizers only focus on performance without considering code size. `RollBin` works on binary level and can also trade-off between performance and code size. Safe ICF [38] is a binary-level technique focusing on code size which folds identical functions. It saves code size with a safe option while keeping the run-time performance of the optimized binaries. With performance issues also taken into account, our approach focuses on unrolled loops, which are common structures in almost all programs. `SubDDG` [18] and `SuffixTree` [35] are the de-optimizations of the loop unrolling at binary level, as we discussed in Section 2.2. They can only accommodate loops with specific patterns, while `RollBin` addresses their limitations and is more flexible to handle the common cases.

7 Conclusion

While often overlooked, the continuously increasing code size can be a first-order constraint on almost all computing platforms. Aiming to alleviate the size issue in a general fashion, this paper presents a `RollBin` to reroll loops at binary level. Through using memory address sequence and customized data dependency analysis, `RollBin` is capable to recognize a greater amount of loops. Experiments on benchmarks and practical applications show that the design effectively reduces code size, significantly outperforming prior arts. Moreover, owing to additional considerations on loop hotness, `RollBin` enables to trade-off between size reduction and performance of rerolling.

Acknowledgments

We would like to thank Yue Weng and the anonymous reviewers for their helpful suggestions on improving this paper. This research was supported by the National Key R&D Program of China (Grant No.2021YFB0301300), the National Natural Science Foundation of China-#U1811461, the Major Program of Guangdong Basic and Applied Research-#2019B030302002, the Guangdong Natural Science Foundation-#2018B030312002, and the Program for Guangdong Introducing Innovative and Entrepreneurial Teams-#2016ZT06D211.

References

[1] David Callahan, Jack J. Dongarra, and David Levine. 1988. Vectorizing Compilers: A Test Suite and Results. In *Proceedings Supercomputing '88*. IEEE Computer Society, 98–105. https://doi.org/10.1109/SUPERC.1988.44642

[2] Milind Chabbi, Jin Lin, and Raj Barik. 2021. An Experience with Code-Size Optimization for Production iOS Mobile Applications. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 363–377. https://doi.org/10.1109/CGO51591.2021.9370306

[3] John Cocke. 1970. Global Common Subexpression Elimination. In *Proceedings of a Symposium on Compiler Optimization*. ACM, 20–24. https://doi.org/10.1145/800028.808480

[4] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. 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)*. ACM, 1–9. https://doi.org/10.1145/314403.314414

[5] Standard Performance Evaluation Corporation. 2021. SPEC CPU® 2017. https://www.spec.org/cpu2017/

[6] Joe Curley and Sanjiv Shah. 2022. oneAPI – The Cross-Architecture, Multi-Vendor Path to Accelerated Computing. https://www.intel.com/content/www/us/en/developer/articles/technical/cross-architecture-multi-vendor-path-computing.html

[7] Anderson Faustino da Silva, Bernardo N. B. de Lima, and Fernando Magno Quintão Pereira. 2021. Exploring the Space of Optimization Sequences For Code-Size Reduction: Insights and Tools. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)*. ACM, 47–58. https://doi.org/10.1145/3446804.3446849

[8] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 378–390. https://doi.org/10.1109/CGO51591.2021.9370322

[9] Thaís Damásio, Vinícius Pacheco, Fabrício Goes, Fernando Pereira, and Rodrigo Rocha. 2021. Inlining for Code Size Reduction. In *Proceedings of the 25th Brazilian Symposium on Programming Languages (SBLP)*. ACM, 17–24. https://doi.org/10.1145/3475061.3475081

[10] Jack J. Dongarra and A. R. Hinds. 1979. Unrolling Loops in FORTRAN. *Softw. Pract. Exp.* 9, 3 (1979), 219–226. https://doi.org/10.1002/spe.4380090307

[11] Kavon Farvardin. 2019. Halo: Wholly Adaptive LLVM Optimizer. https://github.com/halo-project/halo/blob/master/docs/proposal.pdf

[12] Gianluca Frison. 2022. BLASFEO - BLAS For Embedded Optimization. https://github.com/giaf/blasfeo/blob/master/CMakeLists.txt

[13] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2018. BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization. *ACM Trans. Math. Softw.* 44, 4 (2018), 42:1–42:30. https://doi.org/10.1145/3210754

[14] Google. 2022. Android Developers – Reduce Your App Size. https://developer.android.com/topic/performance/reduce-apk-size

[15] Matthew R. Guthaus, Jeff Ringenberg, Dan Ernst, Todd M. Austin, Trevor N. Mudge, and Richard B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization (WWC-4)*. IEEE Computer Society, USA, 3–14. https://doi.org/10.1109/WWC.2001.990739

[16] Todd T. Hahn, Eric Stotzer, Dineel Sule, and Mike Asal. 2008. Compilation Strategies for Reducing Code Size on a VLIW Processor with Variable Length Instructions. In *Proceedings of the High Performance Embedded Architectures and Compilers, Third International Conference (HiPEAC)*, Vol. 4917. Springer, 147–160. https://doi.org/10.1007/978-3-

540-77560-7_11

[17] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (2006), 1–17. https://doi.org/10.1145/1186736.1186737

[18] Erh-Wen Hu, Bogong Su, and Jian Wang. 2016. Instruction Level Loop De-optimization. In *Computer and Information Science 2015*. Springer International Publishing, Cham, 221–234.

[19] Wen-mei W. Hwu and Pohua P. Chang. 1989. Inline Function Expansion for Compiling C Programs. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 246–257. https://doi.org/10.1145/73141.74840

[20] Apple Inc. 2022. Building a Universal macOS Binary. https://developer.apple.com/documentation/apple-silicon/building-a-universal-macos-binary

[21] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 147–158. https://doi.org/10.1145/178243.178256

[22] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. *See* `http://llvm.cs.uiuc.edu`..

[23] LLVM. 2020. MergeFunctions pass, how it works. https://llvm.org/docs/MergeFunctions.html

[24] David Monniaux and Cyril Six. 2021. Simple, Light, Yet Formally Verified, Global Common Subexpression Elimination and Loop-Invariant Code Motion. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 85–96. https://doi.org/10.1145/3461648.3463850

[25] Kateryna Muts, Arno Luppold, and Heiko Falk. 2019. Compiler-Based Code Compression for Hard Real-Time Systems. In *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 72–81. https://doi.org/10.1145/3323439.3323976

[26] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia Lago. 2018. An Extensible Approach for Taming the Challenges of JavaScript Dead Code Elimination. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 391–401. https://doi.org/10.1109/SANER.2018.8330226

[27] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14. https://doi.org/10.1109/CGO.2019.8661201

[28] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2014. Dynamic Reconstruction of Relocation Information for Stripped Binaries. In *International Workshop on Recent Advances in Intrusion Detection*, Vol. 8688. Springer, 68–87. https://doi.org/10.1007/978-3-319-11379-1_4

[29] Daejin Park, Min-Woo Jung, and Jeonghun Cho. 2017. Area Efficient Remote Code Execution Platform With On-Demand Instruction Manager For Cloud-Connected Code Executable IoT Devices. *Simul. Model. Pract. Theory* 77 (2017), 379–389. https://doi.org/10.1016/j.simpat.2016.08.010

[30] Matteo Perotti, Pasquale D Schiavone, Giuseppe Tagliavini, Davide Rossi, Tariq Kurd, Mark Hill, Liu Yingying, and Luca Benini. 2020. HW/SW Approaches for RISC-V Code Size Reduction. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*. 1–7. https://doi.org/10.3929/ethz-b-000461404

[31] Rodrigo C. O. Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael F. P. O'Boyle. 2022. Loop Rolling for Code Size Reduction. In *Proceedings of the 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 217–229. https://doi.org/10.1109/CGO53902.2022.9741256

[32] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim M. Hazelwood, and Hugh Leather. 2021. HyFM: Function Merging for Free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 110–121. https://doi.org/10.1145/3461648.3463852

[33] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function Merging by Sequence Alignment. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 149–163. https://doi.org/10.1109/CGO.2019.8661174

[34] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective Function Merging in the SSA Form. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 854–868. https://doi.org/10.1145/3385412.3386030

[35] Greg Stiff and Frank Vahid. 2005. New Decompilation Techniques for Binary-Level Co-Processor Generation. In *Proceedings of the 2005 International Conference on Computer-Aided Design (ICCAD)*. IEEE Computer Society, 547–554. https://doi.org/10.1109/ICCAD.2005.1560127

[36] Sean Stirling, Rocha Rodrigo C. O., Kim M. Hazelwood, Hugh Leather, Michael F. P. O'Boyle, and Pavlos Petoumenos. 2022. F3M: Fast Focused Function Merging. In *Proceedings of the 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 242–253. https://doi.org/10.1109/CGO53902.2022.9741269

[37] Sriraman Tallam. 2019. Propeller: Profile Guided Optimizing Large Dcale LLVM-Based Relinker. https://github.com/google/llvm-propeller/blob/plo-dev/Propeller_RFC.pdf

[38] Sriraman Tallam, Cary Coutant, Ian Lance Taylor, Xinliang David Li, and Chris Demetriou. 2010. Safe ICF: Pointer Safe and Unwinding Aware Identical Code Folding in Gold. In *GCC Developers Summit*. http://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=view&target=tallam.pdf

[39] TensorFlow. 2021. Build TensorFlow Lite with CMake. https://www.tensorflow.org/lite/guide/build_cmake

[40] TensorFlow. 2022. ML for Mobile and Edge Devices - TensorFlow Lite. https://www.tensorflow.org/lite

[41] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and Exploiting Optimal Function Inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 977–989. https://doi.org/10.1145/3503222.3507744

[42] Tobias J. K. Edler von Koch, Igor Böhm, and Björn Franke. 2010. Integrated Instruction Selection and Register Allocation For Compact Code Generation Exploiting Freeform Mixing of 16- and 32-bit Instructions. In *Proceedings of the 8th International Symposium on Code Generation and Optimization (CGO)*. ACM, 180–189. https://doi.org/10.1145/1772954.1772980

[43] Tobias J. K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. 2014. Exploiting Function Similarity For Code Size Reduction. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*. ACM, 85–94. https://doi.org/10.1145/2597809.2597811

[44] Andrew Wolfe and Alex Chanin. 1992. Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*. ACM / IEEE Computer Society, 81–91. https://doi.org/10.1109/MICRO.1992.697002

[45] Xianhong Xu, Simon Jones, and Christopher T. Clarke. 2003. ARM/THUMB Code Compression for Embedded Systems. In *Proceedings of the 12th IEEE International Conference on Fuzzy Systems*. 32–35. https://doi.org/10.1109/ICM.2003.238300

[46] Ruoyu Zhou and Timothy M. Jones. 2019. Janus: Statically-Driven and Profile-Guided Automatic Dynamic Binary Parallelisation. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 15–25. https://doi.org/10.1109/CGO.2019.8661196