Loop Rolling for Code Size Reduction

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

Pavlos Petoumenos
University of Manchester, UK
pavlos.petoumenos@manchester.ac.uk

Björn Franke
University of Edinburgh, UK
bfranke@inf.ed.ac.uk

Pramod Bhatotia
Technical University of Munich
pramod.bhatotia@in.tum.de

Michael O'Boyle
University of Edinburgh, UK
mob@inf.ed.ac.uk

***Abstract*—Code size is critical for resource-constrained devices, where memory and storage are limited. Compilers, therefore, should offer optimizations aimed at reducing code size. One such optimization is loop rerolling, which transforms a partially unrolled loop into a fully rolled one. However, existing techniques are limited and rarely applicable to real-world programs. They are incapable of handling either partial rerolling or straight-line code.**

In this paper, we propose RoLAG, a novel code-size optimization that creates loops out of straight-line code. It identifies isomorphic code by aligning SSA graphs in a bottom-up fashion. The aligned code is later rolled into a loop. In addition, we propose several optimizations that increase the amount of aligned code by identifying specific patterns of code. Finally, an analysis is used to estimate the profitability of the rolled loop before deciding which version should be kept in the code.

Our evaluation of RoLAG on full programs from MiBench and SPEC 2017 show absolute reductions of up to 88 KB while LLVM's technique is hardly distinguishable from the baseline with no rerolling. Finally, our results show that RoLAG is highly applicable to real-world code extracted from popular GitHub repositories. RoLAG is triggered several orders of magnitude more often than LLVM's rerolling, resulting in meaningful reductions on real-world functions.

***Index Terms*—Code-Size Reduction, Loop Optimization, Loop Rerolling, LLVM, Compiler Optimization**

I. Introduction

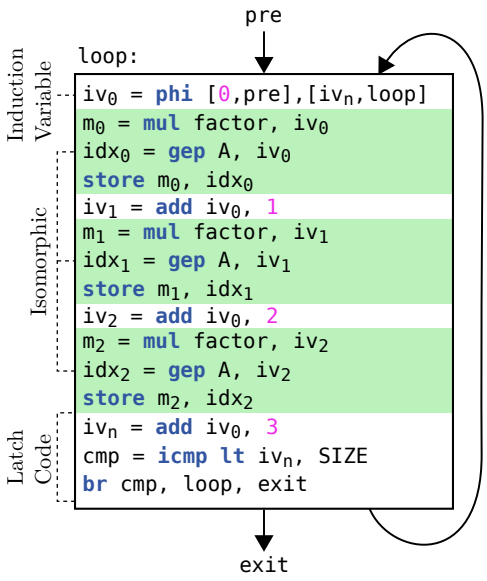
Code size is critical for programs targeting resource-constrained systems. Within embedded systems, the memory occupied by code is a significant issue. Small differences translate directly to increases in cost, leading to lost profit at large scales [1], [2]. The size of mobile applications is important not only for the end-user experience but also for vendors as download size can greatly impact their revenues [3]. In such scenarios, reducing the application footprint is a priority.

Compilers currently provide specific optimizations tailored for code size reduction [4], [2], [5]. These include common sub-expression elimination [6], partial redundancy elimination [7], constant propagation [8], dead and unreachable code elimination [9]. Modern size optimizations have been more ambitious, focusing on merging equivalent code to avoid duplicates. Function merging identifies functions with sufficient similarity and merges them into a single function [10], [2], [11], [12]. Function outlining is another such optimization that works at the block level. It identifies equivalent basic blocks across all the functions and extracts them into a function, replacing all the copies by a function call [3].

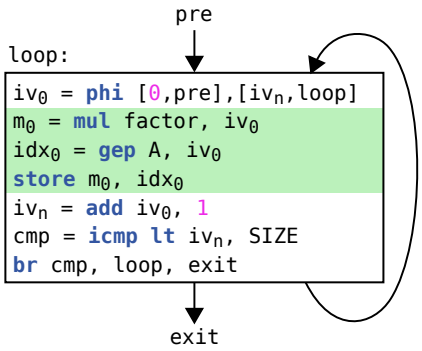
Once we have all similar code merged, is there anything else we can do to compress code size? One promising approach is loop (re)rolling which works at the instruction level. It works by transforming equivalent instructions from a single block into a loop. However, existing loop rerolling is limited to transforming partially unrolled loops into fully rolled loops. It is incapable of rolling straight-line code into loops. This highly restrictive approach is rarely applicable to real-world code. Our insight is that the weak results of the existing loop rerolling implementations are not due to the lack of duplicate code but due to the rigid and overly restrictive algorithms they use to find code duplicates.

In this paper, we propose RoLAG, a novel loop rolling technique for straight-line code. We use the term *loop rolling* to refer to our technique because we are not specifically focused on reversing a prior unrolling transformation, since code similarity can arise in different forms [10], [2], [3]. Our technique was inspired by the Superword Level Parallelism (SLP), which is a vectorizer for straight-line code [13], [14]. RoLAG identifies isomorphic code by aligning SSA graphs in a bottom-up fashion. It starts by scanning the code, collecting and grouping instructions that are likely to lead into isomorphic code. Then we expand these groups of seed instructions by following their use-def chains, aligning nodes from their SSA graphs in a data-structure we call the *alignment graph*. The alignment graph describes pieces of code that are equivalent and therefore can be rolled into a loop. We also propose several optimizations that tend to increase the amount of aligned code by identifying specific patterns of code. Finally, we use a profitability analysis to decide which version should be kept.

Our evaluation on real-world functions extracted from popular GitHub repositories, namely, the AnghaBench suite, shows that RoLAG is triggered several orders of magnitude more often than LLVM's rerolling technique, producing meaningful reductions. We have also evaluated on full programs from MiBench and SPEC 2017 benchmark suites, where RoLAG is also triggered more often than the alternative. However, on full programs, reduction is limited to only a few KBs and a relative reduction of less than 3%. Finally, we provide a detailed analysis of both techniques on the TSVC benchmark



(a) Original loop.



(b) Loop after rerolling.

Fig. 1: Example demonstrating LLVM's loop rerolling technique on a partially unrolled loop.

suite, where we expose their limitations and suggest future improvements.

In this paper, we make the following contributions:

- We propose a novel loop rolling technique for straight-line code based on a bottom-up graph alignment solution.
- We describe how we can identify and represent several patterns of code in the rolled loop.
- We offer a powerful scheduling analysis that enables our loop rolling to go beyond contiguous sequences of repeating code, allowing instructions to be rearranged.
- We show the effectiveness of our technique on real-world code.

II. Background

In this section, we describe the existing loop rerolling strategy implemented in LLVM. This implementation is under development since 2013 and with contributions from a total of 39 different developers [15]. The aim of the pass is essentially to revert previous partial unrolls of single-block loops.

For each single-block loop, LLVM first looks for a basic induction variable of integer type. A basic induction variable is a variable that is either incremented or decremented by the same loop-invariant value at each iteration of the loop [16]. In the *static single assignment* (SSA) form, where names are defined only once, the recursive aspect of a basic induction variable is denoted using a phi-node. In the example from Figure 1a, `iv0` is an induction variable because it has an incoming value of zero which defines its starting value at the first iteration and another incoming value that defines its recursive increment by 3 performed at each iteration.

Such induction variables serve as starting points for LLVM's loop rerolling strategy. Following *definition-use chains*, it searches for the instructions representing unrolled increments of the induction variable. If this basic block was indeed generated through unrolling, the induction variable and the increment instructions will define the start of all unrolled iterations. So for each one of these *root* instructions, LLVM collects all instructions in the graph following their definition-use chains, except for instructions in the *latch code*. For the example of Figure 1a, this process leads us to Figure 2 which shows the induction variable with its two increment instructions and the three green blocks containing the instructions collected for each root.

These sets of instructions must have equivalent instructions representing each one of the unrolled iterations. For simplicity, these instructions are collected and analyzed in the order they appear in their basic block. Every one of these groups must have the same number of instructions and corresponding instructions must have the same opcode and data types. The operands of two equivalent instructions are either the same, representing loop-invariant values, or other equivalent instructions that have been previously analyzed.

At the end of this process, all instructions in loop must have been covered, i.e., the loop contains only the root instructions, the equivalent pieces of code, and the latch code. If all these constraints are satisfied, the loop is rerolled, as shown in Figure 1b.

III. Motivation

Even though this optimization has been part of LLVM for more than eight years, more often than not it fails to produce any noticeable code size reduction. Two such cases, both extracted from the Linux kernel, are shown in Figures 3 and 4. All major compilers fail to optimize these two examples. LLVM's loop rerolling technique is unable to optimize them since it is limited to loops that have only been partially unrolled and contain simple data dependencies.

Figure 3a shows a function [17] consisting of five call statements, all of them to the same function, `vst1q_u8`. Although their operands are not completely identical, they follow a regular pattern that can be reconstructed when rolled into a loop, as shown in Figure 3b. The induction variable, ranging from 0 to 4, is used to reconstruct the list of operands of each call statement. The first operand for the first iteration (`state + 0*16`) is not the same as in the original function (`state`) from a naive programmatic point of view but they are equivalent in terms of result. Manually rolling this code into

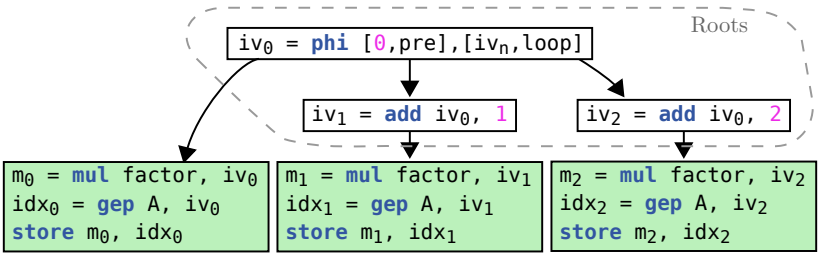


Fig. 2: Data structure created by the existing technique to identify loop rerolling opportunities.

```
static void aegis128_save_state_neon(
  struct aegis128_state st, void *state) {
  vst1q_u8( state     , st.v[0] );
  vst1q_u8( state + 16, st.v[1] );
  vst1q_u8( state + 32, st.v[2] );
  vst1q_u8( state + 48, st.v[3] );
  vst1q_u8( state + 64, st.v[4] );
}
```

(a) Original function found in the Linux code base.

```
static void aegis128_save_state_neon(
  struct aegis128_state st, void *state) {
  for (int i = 0; i<=4; i++)
    vst1q_u8(state + i*16, st.v[i]);
}
```

(b) Manually optimized version of the function.

Fig. 3: Example of a function with a sequence of calls that could be rolled into a loop, reducing code size by about 20%.

```
struct hdmi_audio_format {
  int sample_size; int samples_word;
  int sample_order; int justification;
  int type; int en_sig_blk; ... } ;

u32 r = hdmi_read_reg(wp->base,
        HDMI_AUDIO_CFG);
r = FLD_MOD(r, fmt->en_sig_blk,   5, 5);
r = FLD_MOD(r, fmt->type,         4, 4);
r = FLD_MOD(r, fmt->justification, 3, 3);
r = FLD_MOD(r, fmt->sample_order, 2, 2);
r = FLD_MOD(r, fmt->samples_word, 1, 1);
r = FLD_MOD(r, fmt->sample_size,  0, 0);
```

(a) Original function found in the Linux code base.

```
u32 r = hdmi_read_reg(wp->base,
        HDMI_AUDIO_CFG);
int *ptr = &(aud_fmt->sample_size);
for (int i = 5; i>=0; i--)
  r = FLD_MOD(r, ptr[i], i, i);
```

(b) Manually optimized version of the function.

Fig. 4: Example extracted from the Linux repository. The sequence of chained function calls could be rolled into a loop.

a loop reduces its size in the final object file from 143 bytes down to only 115 bytes, a reduction of about 20%. However, existing compilers fail to optimize this code, keeping it in its straight-line form with five replicas of the same function call[1].

Figure 4a shows a more complex example [18]. This function contains a chain of six function calls where each value returned from one call is passed as argument to the next one. While this complicates the analysis, it is still possible to roll this into a loop as shown in Figure 4b. We first transform the chained pattern into a *loop-carried dependence* [19] and then we treat the elements from the data structure named `hdmi_audio_format` as an array of integers that are accessed in reverse in consecutive iterations. In the final object file, the original version of this function takes a total of 317 bytes while the rerolled version takes only 274 bytes, resulting in a reduction of about 13.6%.

These examples show that existing compilers fail to identify significant opportunities for code size reduction. Existing loop rerolling techniques only work for partially unrolled loops that can be perfectly rerolled into a loop and cannot handle straight-line code. In this paper, we propose a novel solution which removes such superfluous constraints and is capable of transforming arbitrary pieces of equivalent code into a loop, as long as this is profitable.

IV. Loop Rolling

In this paper, we propose RoLAG, a novel loop rolling technique based on graph alignment that works on straight-line code. Our technique works in a bottom-up fashion and consists of five major stages: seed collection, graph alignment, scheduling analysis, code generation, and profitability analysis. Figure 5 shows the main procedure responsible for processing each basic block.

First, RoLAG scans all instructions in a basic block, identifying groups of instructions which are likely to lead to *isomorphic* code. These instructions, which we call seeds, are usually stores, function calls, or instructions that may form reduction trees. Starting from each group of seed instructions, we follow their *use-def chains* in order to identify similarities in their SSA graphs. As a result of this bottom-up traversal, we construct an alignment graph. The alignment graph represents a correspondence relation among the input SSA graphs, distinguishing between matching and mismatching nodes. Matching nodes correspond to either isomorphic instructions or identical values while mismatching nodes represent where the input SSA graphs differ. We further improve upon these two types of nodes with special nodes that represent higher-order code patterns.

Once the alignment graph is built, the scheduling analysis verifies whether or not the instructions from the alignment

[1]https://godbolt.org/z/vqaoKo4rT

```
RollLoop(BB):
  SG = collectSeedGroups(BB)
  while not SG.empty():
    S = SG.pop()
    SB = S.getBlock()
    AG = AlignmentGraph(S)
    valid = analyzeScheduling(SB, AG)
    if valid:
      L = generateLoop(SB, AG)
      if size(L) < size(SB):
        replace(SB,L)
      else:
        delete(L)
```

Fig. 5: Pseudo code of the main procedure in RoLAG. This procedure repeats for each basic block in the program.

graph can be rearranged as loop iterations while still preserving its original semantics. If successful, RoLAG generates the rolled loop from the alignment graph. The number of input SSA graphs used to build the alignment graph represents the number of iterations of the rolled loop. Matching nodes can be immediately represented inside the rolled loop while mismatching nodes must be handled externally and used as input values to the rolled loop. Since there are code size overheads for handling mismatching nodes and external uses, loop rolling may also lead to code size increase. Therefore, we need a profitability analysis to identify when loop rolling can reduce code size. We estimate the size of both the original code and the rolled loop with a cost model that estimates the size of individual IR instructions when lowered to the target binary. The version with the smallest size is kept.

A. Seed Collection

RoLAG scans each basic block collecting seed instructions, which consists mainly of store instructions, function calls, and instructions that may lead to reduction trees. These seed instructions are then grouped based on some similarity aspects. The store instructions are grouped based on their data type and base address. The function calls are grouped based on their callee. These groups of seed instructions will serve as the starting point for our graph alignment strategy. Instructions that may lead to reduction trees are processed separately, as discussed in Section IV-C5.

B. Aligning Isomorphic Code

RoLAG constructs an *alignment graph* for each group of seed instructions. Figure 7 shows an alignment graph constructed out of three input SSA graphs. Each node in the alignment graph represents a group of instructions or values from the input SSA graphs. In the simplest case, nodes are either classified as matching or mismatching. A matching node contains a group of equivalent instructions or identical values that can be merged using a common representation inside the rolled loop. Mismatching nodes have groups of instructions or values that cannot be merged into the rolled loop.

```
AlignmentGraph::build(GV):
  N = createNode(GV)
  add(N)
  for i in range(N.numOperands()):
    GO = N.getOperandGroup(i)
    N.addChild(i, build(GO))
  return N
```

Fig. 6: Pseudo code of the main procedure used for building an alignment graph. It starts with the group of seed instructions, growing recursively towards the operands.

Instructions are equivalent if: (1) they have the same opcode; (2) they have equivalent types; and (3) their corresponding operands have equivalent types. Similarly, types are equivalent if they can be bitcast in a lossless way from one to the other. Type equivalence for function calls means that they have identical function types, i.e. identical return types and identical list of parameters.

The alignment graph is built by following the SSA graph in a bottom-up fashion, i.e., following the *use-def* chains, starting from the seed instructions and growing towards their operands. Once a node is created from a group of instructions, their corresponding operands are grouped and the algorithm proceeds recursively with each one of these groups. This bottom-up expansion stops at nodes that contain only constants, function parameters, or mismatching values.

Figure 7b shows a simple alignment graph built out of the SSA graphs shown in Figure 7a. This simple alignment graph contains only the matching and mismatching nodes described above. Even though this approach is capable of identifying and aligning isomorphic subgraphs, in practice this is still very limited. We can reduce the number of mismatching nodes by identifying specific patterns of code. For example, in Figure 7b, although the second operand of the `gep` node is a mismatching node, since all its values are different, it is clear that they follow a simple pattern.

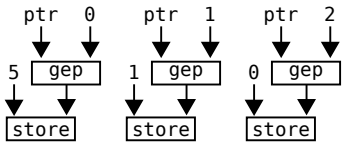
C. Abstracting Special Code Patterns

In this subsection, we explore several patterns of code that can be leveraged to increase the amount of aligned code.

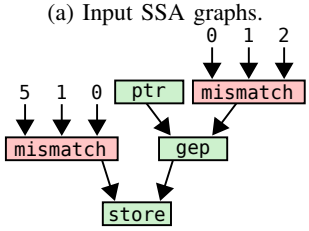
1) Monotonic Integer Sequences: While building an alignment graph, we often encounter nodes containing only integer constants. The most trivial case is when all constants in the sequence are identical. Representing such a sequence of identical constants in each iteration of the rolled loop is trivial since we can directly use the constant itself. However, we can go beyond sequences of only identical constants.

Figure 8 shows a monotonically increasing sequence. This sequence follows a regular pattern because the difference from one number to the next is always the same. Hence, we can represent this sequence as a function of the main induction variable of the rolled loop.

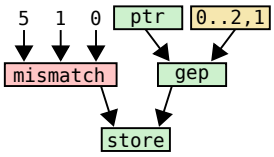
We can formally describe such a sequence as follows: Given a sequence of integer constants $(S_0, S_1, \ldots, S_n)$, with $n \in \mathcal{N}$, if $S_i - S_{i-1}$ equals $S_1 - S_0$ for all $0 < i \leq n$, then $S_i =$



(a) Input SSA graphs.



(b) Simple alignment graph.



(c) Improved alignment graph.

Fig. 7: The alignment graph built bottom-up from the seed instructions. We highlight exact matches in green and mismatches in red. Special nodes are highlighted in yellow.

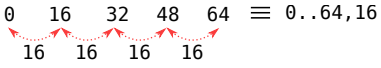


Fig. 8: Example of a monotonically increasing sequence that grows by a fixed value.

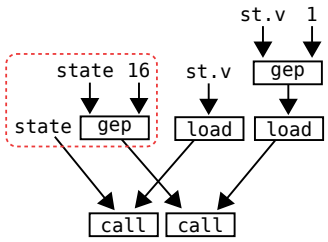
$S_0 + i \cdot (S_1 - S_0)$. In other words, we can represent the i-th element of the given sequence as a linear function of i. Note that the sequence can be either monotonically increasing or decreasing. In this paper, we denote these sequences by S_0 .. S_n,$(S_1 - S_0)$, as shown in Figure 8.

Figure 7 shows a particular case where we have a sequence of integers that differ by one. We can use the rule discussed above to improve the alignment, as shown in Figure 7c.

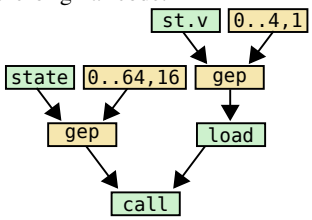
2) Neutral Pointer Operations: Figure 9a shows two out of the five SSA graphs from the example in Figure 3. For this example, we can build the alignment graph shown in Figure 9b. However, we cannot directly match the subgraphs highlighted in Figure 9a.

The highlighted pattern often arise when performing pointer operations. The `getelementptr` instruction (shortened as `gep`), offsets the input pointer returning another pointer. The offset of a pointer by zero is equivalent to the pointer itself. Therefore, we can leverage this equivalence to make the two highlighted subgraphs isomorphic.

While building a node in the alignment graph, RoLAG identifies this pattern by scanning though the group of instructions and values. If all `getelementptr` instructions in this group have the same pointer operand and all the remaining values in the group are equal to this pointer, then we can use the equivalence rule. Figure 9 shows how this equivalence rule is applied in a real example.



(a) Two of the SSA graphs from the original code.



(b) The alignment graph.

Fig. 9: Loop rolling of the example in Figure 3a. This example illustrates how neutral pointer operations can be exploited to improve graph alignment.

3) Algebraic Properties of Binary Operations: In this paper, we also exploit algebraic properties of binary operations in order to improve the graph alignment strategy.

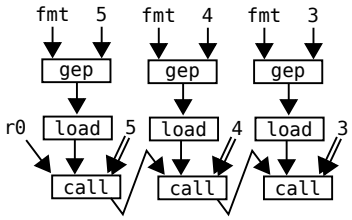
First, we exploit the neutral element of binary operations. For example, the neutral element of the addition is zero, i.e., the operation $a + 0$ equals to a. We can leverage this identity in a similar way to what has been described for neutral pointer operations. Given a sequence of instructions and values, RoLAG identifies the most frequent binary operation in the sequence. If a binary operator was found and this operator has a neutral element, then RoLAG uses the identity on the mismatching values in the sequence. The mismatching values can be anything, even another binary operation, as long as they have different opcodes to the main binary operation.

Second, we exploit the commutativity property of certain binary operations. While grouping the operands from commutativity operations, instead of simply grouping them based on their position, we can group them by maximizing their similarity, uncovering more profitable alignment.

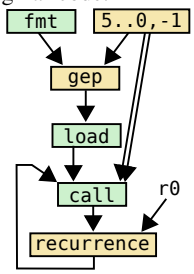
4) Chained Dependences: Figure 10a shows three out of the six SSA graphs from the example in Figure 4. Note how these SSA graphs are chained together with one call instruction being the input for the next. For this particular example, we can build the alignment graph shown in Figure 10b.

The construction of the alignment graph starts with the function calls, which are the seed instructions. Let $(I_0, I_1, \ldots, I_5)$ be the seed instructions. Once the matching node is created for the seed instructions, we proceed towards their operands. The group of the first operand of each call is $(r0, I_0, I_1, \ldots, I_4)$. This chained dependence in the SSA graph translates to a cyclic pattern in the alignment graph, where we create a special node that recursively feeds into itself.

In the rolled loop, the *recurrence* node is lowered directly to



(a) Two of the SSA graphs from the original code.



(b) The alignment graph.

Fig. 10: Loop rolling of the example in Figure 4. The recurrence node is used to represent the chained sequence of calls.

a *phi-node*. For this example, the initial value of the phi-node is the value `r0` and the function call is the recurrence value of the phi-node.

5) Reduction Trees: A reduction tree is a tree where all its internal nodes perform the same associative binary operation [20], [21], [22], [23]. The associative property allows us to rearrange the reduction tree in different ways while preserving its semantics. Therefore, we can leverage this property to roll the reduction tree into a loop. Similar to other optimizations, re-association of floating-point instructions must be explicitly enabled, e.g., via the compiler's `fast-math` flag.

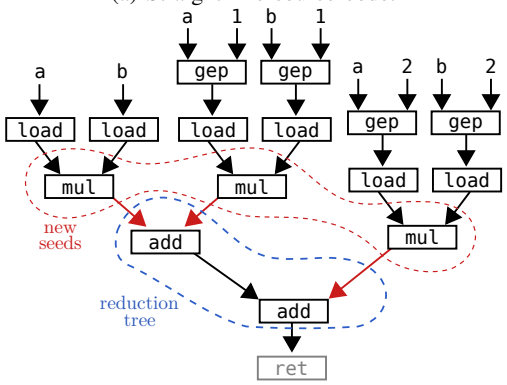
Figure 11 shows an example that contains a reduction tree. Unlike the other cases, in the presence of a reduction tree, the alignment graph grows out of a single seed instruction. RoLAG starts by collecting all the internal nodes of the reduction tree, as highlighted in Figure 11b. Then, a single special node is created in the alignment graph in order to represent the whole reduction tree. This special node becomes the root of the alignment graph. Afterwards, all the incoming nodes become a new group of seeds, as highlighted in Figure 11b. The construction of the alignment graph continues with this group of seeds following the standard procedure previously described. The final alignment graph is shown in Figure 11c.

In the rolled loop, the reduction node is lowered to a single instance of the binary operation followed by a *phi-node*. The *phi-node* acts as an accumulator and its initial value is the neutral element of the binary operation involved in the reduction. This implementation follows from the associativity property and is the idiomatic representation of loop-based reductions in the SSA form [24], [21].

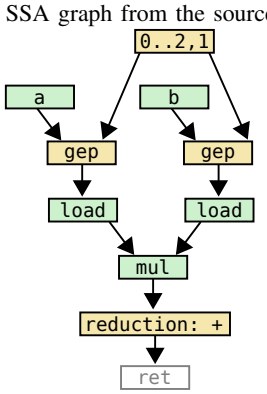
6) Joining Alignment Graphs: We may often find a piece of code that contains a sequence of alternating patterns. In this case, we need to be able to roll these patterns of code into a single loop. For example, Figure 12a shows a sequence

```
int DotProduct( const int *a, const int *b ){
  return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}
```

(a) Straight-line source code.



(b) The SSA graph from the source code.



(c) The alignment graph.

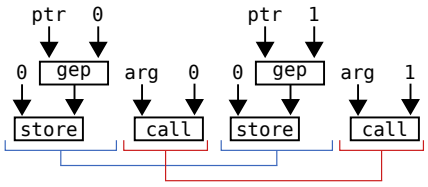
Fig. 11: Example of a reduction tree. The whole reduction tree is represented by a special node in the alignment graph.

with two alternating patterns of code. The first one has store instructions as its seeds and the second has a group of function calls as its seeds. Rolling these two patterns of code into a single loop is not only preferable, with respect to code size, but also necessary. As we explain in Section IV-D, because these instructions write to memory or may have side effects, we cannot rearrange them as two separate loops. By rolling them into a single loop, we preserve the alternating aspect of this sequence.

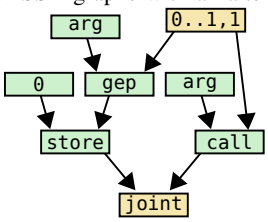
Given a group of seed instructions and their position in the basic block, we search for other groups of seed instructions that alternate in position with the seeds from the first group. All groups must have the same number of seed instructions in order to be joined together. Once the joint node has been created, the alignment-graph construction algorithm proceeds towards each group of seeds involved in the creation of the joint node. Unlike the other special nodes, the joint node generates no code, it only specifies the order the joined aligned graphs must appear inside the rolled loop.

D. Scheduling Analysis

Once it builds the alignment graph, RoLAG analyzes whether it can rearrange the instructions inside the basic



(a) Sequence of SSA graphs with an alternating pattern.



(b) The alignment graph.

Fig. 12: Example of two groups of seed instructions combined in a single alignment graph by a *joint* node.

block, grouping the instructions from the alignment graph as a contiguous sequence, while still preserving the program semantics. The goal is to prove that the instructions can be statically rearranged in the same order they would be executed if they were rolled into a loop.

Figure 13 depicts the reordering requirements imposed by the loop rolling transformation. In summary, all instructions from mismatching nodes (labeled MM-#), as well as their dependences, must precede the instructions from the matching nodes (labeled M-#). All instructions outside of the alignment graph (labeled I-#) must be placed either before or after those from the matching nodes. Instructions independent from the alignment graph (I2, I-3, and I-5) are placed after those from the matching nodes while instructions that are depended on by the alignment graph (I-1 and I-4) are placed before them. Finally, it must be possible to order the instructions from the matching nodes (labeled M-#) following the execution order of each iteration of the loop.

These rearrangements are prohibited if two instructions with potential side effects, including instructions that access memory, need to switch their execution order. Simple instructions, such as casting or binary operations, are allowed to cross any other instruction as long as there is no dependence between them. The loop rolling transformation allows for instructions inside the alignment graph to depend on instructions outside it, e.g., the mismatching nodes and their operands. The transformation also allows for instructions outside the alignment graph to depend on instructions inside it, where arrays are used to store the output values, as explained in Section IV-E. Finally, it also allows for circular dependences inside the alignment graph, as explained in Section IV-C4. However, it prohibits circular dependences that cross the boundaries of the alignment graph since one of the dependences would be broken by placing the external instructions either before or after the loop.

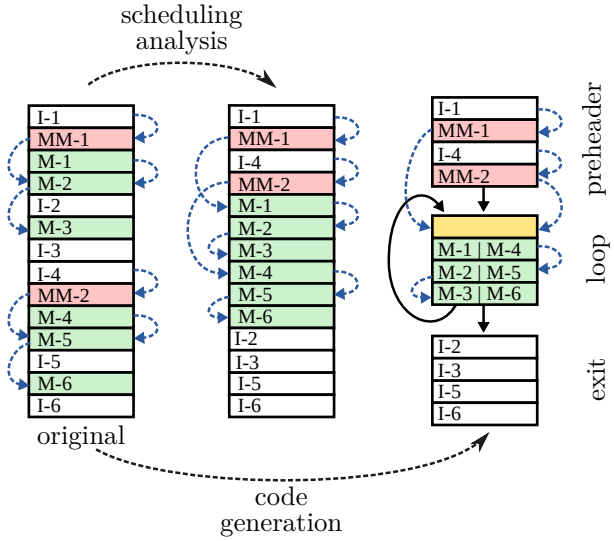


Fig. 13: Scheduling analysis guarantees correctness.

E. Code Generation

The code generator takes the alignment graph as input. Figure 14 illustrates how the rolled loop is generated from the alignment graph. First it generates the basic blocks and the control instructions of the loop. The loop count is the number of seed instructions (or SSA graphs) used to build the alignment graph. In the presence of a reduction tree, the loop count is the number of seed instructions created after the reduction node.

The basic blocks generated are: (1) The *pre-header*, which precedes the loop. It also contains the code necessary for handling the mismatching nodes. (2) The single block of the rolled loop itself. (3) The exit block, which succeeds the loop. If there are values computed inside the loop with external uses, the exit block also contains the code for extracting those values from the loop. Meanwhile, the control instructions comprise of an induction variable that ranges from zero up to the loop count, a conditional check for the loop count, and branches for entering and exiting the loop.

Finally, the code from the alignment graph is created in *post-order*, recursively constructing the def-use chains. Matching nodes are simply copied into the loop. Mismatching nodes have all their values stored in arrays which are then extracted using a single load instruction inside the loop. If all the mismatching values are constants, we simply create a global array of constants. Otherwise, we use a stack array and store instructions are created in the pre-header block for each one of the values from the mismatching node. For nodes with external use, i.e., nodes with instructions that are depended on by other instructions outside of the alignment graph, their values are stored into an array at each iteration. External uses need a single store instruction inside the loop block and the used values are extracted with load instructions in the exit block.

F. Profitability Analysis

The costs for handling the mismatching nodes and external uses may often outweigh the benefits of merging code into a

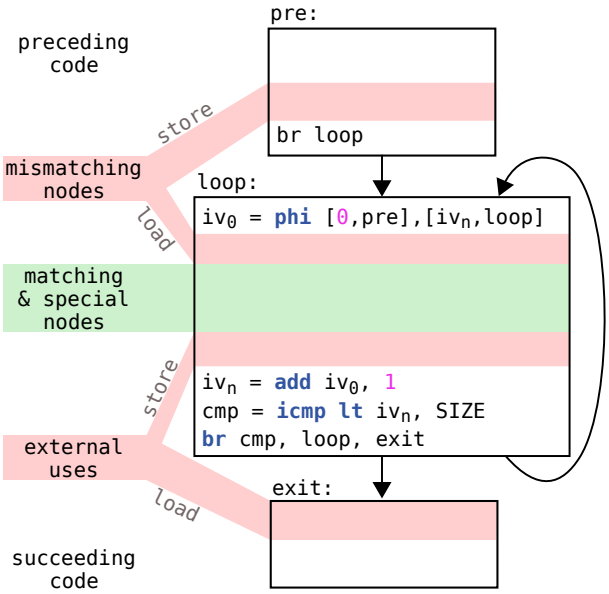


Fig. 14: Diagram of the code generation.

loop. The final decision is based on estimates of the code-size costs for both the rolled loop and the original straight-line code. The version with the smallest estimated code size is chosen.

The profitability is measured with the help of the compiler's target-specific cost model. The cost of each instruction comes from querying this compiler's built-in cost model, which provides a cost estimation that approximates the size of an IR instruction when lowered to the target machine. We use the code-size cost model provided by LLVM's target-transformation interface (TTI), which is used in the decision making of most optimizations [25], [11].

V. Evaluation

In this section, we evaluate our approach in terms of code size reduction. We compare RoLAG with LLVM's loop rerolling technique described in Section II. We evaluate both techniques on four different benchmark suites: AnghaBench [26], MiBench [27], SPEC 2017 [28], and TSVC [29].

We build the benchmark suites with Clang/LLVM using `-Os` for size optimization, with loop unrolling disabled, except for the experiments on the TSVC suite where we manually unroll all inner loops. The baseline in all cases has no loop (re)rolling enabled. We target the Intel x86 architecture, performing all experiments on a quad-core Intel Xeon CPU E5-2650 with 64 GiB of RAM, running Ubuntu 18.04.3 LTS.

A. Code-Size Reduction on Real Code

First, we evaluate the effectiveness of both RoLAG and LLVM's loop rerolling technique on real-world code. For this evaluation, we use the AnghaBench suite [26], which provides one million compilable functions. All these functions were extracted from the most popular GitHub repositories that contain C source files, including well-known repositories such as: PostgreSQL, numpy, Linux, FFmpeg, etc. Our results on AnghaBench show that RoLAG can find a significant number of optimization opportunities in real-world programs, which include the two examples from Section III.

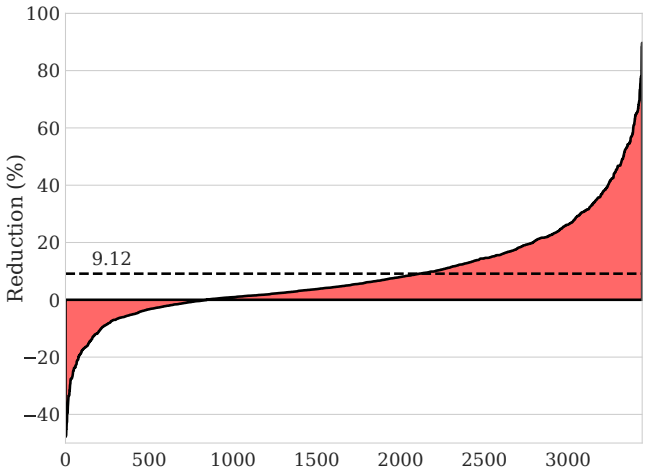


Fig. 15: Code size reduction on the AnghaBench suite. RoLAG achieves an average reduction of 9.12% on the final object file.

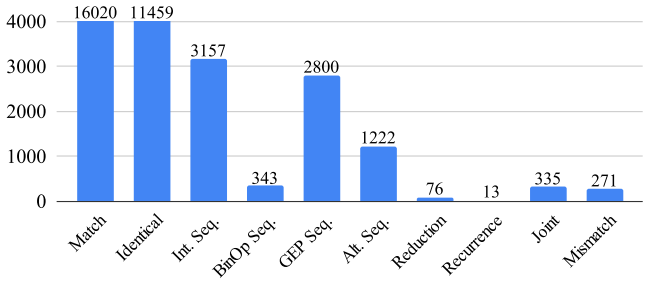


Fig. 16: Breakdown of the different types of node used in profitable alignment graphs across the AnghaBench suite.

Figure 15 shows the code size reduction achieved by RoLAG on AnghaBench. For simplicity, we are only considering functions that were visibly affected by either technique, which is only around 3500 out of the whole AnghaBench. Although this represents a small percentage, RoLAG was still triggered several orders of magnitude more often than LLVM's rerolling. For this reason, this figure omits the results from the LLVM's rerolling because it has a negligible effect on this benchmark suite, affecting less than 50 functions.

The function with the best percentage reduction was extracted from the Linux kernel-based virtual machine [30]. It receives two objects of two different structures, copying several fields from one object to the other. Similar to what we saw in Figure 4, the loop rolling transformation can treat the two objects as arrays and use the induction variable to index the necessary fields from both arrays. RoLAG only needs to make sure that all fields have data types with the same bit size and that they can be properly indexed starting from a base address in the object. Finally, RoLAG converts all 72 memory copy operations into a single loop-based copy operation, reducing the *text* size of the function by almost 90%.

Suite	Program	Binary Size	Reduction		Rolled
		KB	KB	%	Loops
MiBench	typeset	534.4	-0.7	-0.1	8
	sha	3.3	-0.03	-0.8	3
	pgp	179.2	0.02	∼0	5
	gsm	48.6	0.05	0.1	1
	jpeg_d	116.7	0.15	0.1	12
	jpeg_c	121.1	0.23	0.2	12
	ghostscript	908.8	0.66	0.1	68
	tiff2bw	240.1	3.11	1.3	25
	tiff2dither	239.5	3.3	1.4	24
	tiff2median	239.6	3.3	1.4	25
	tiff2rgba	243.8	3.4	1.4	27
SPEC'17	657.xz_s	158.2	-0.4	-0.2	8
	620.omnetpp_s	1512.2	-0.07	∼0	20
	605.mcf_s	17.8	-0.02	-0.1	1
	644.nab_s	149.9	-0.01	∼0	15
	631.deepsjeng_s	68.8	0.09	0.1	7
	619.lbm_s	15.4	0.14	0.9	3
	625.x264_s	392.2	0.27	0.1	86
	638.imagick_s	1574.9	0.95	0.1	73
	511.povray_r	790.8	21.05	2.7	480
	526.blender_r	8508.5	87.91	1.1	2580

TABLE I: Code reductions achieved by RoLAG on programs from both the MiBench and SPEC 2017 benchmark suites.

We also observe a code-size increase in some of the functions, which results from false positives caused by the profitability analysis. These false positives happen for two main reasons: (1) Rolling a straight-line code into a loop may prevent future optimizations and such costs cannot be estimated by a cost model. It is well known that changing the order of the optimization passes can have a huge impact in the final binary. Therefore, choosing at which point in the compilation pipeline loop rolling can be most effective is also an important research topic. (2) Cost models can be inaccurate as they estimate at the IR level the size of individual instructions when lowered to the target architecture. However, this is not a direct mapping and instruction scheduling, register allocation, as well as other optimizations, play a significant role lowering the bitcode to the target binary.

Figure 16 shows a breakdown of the types of nodes from profitable alignment graphs found in AnghaBench. As expected, nodes with matching instructions or identical values are the most common in profitable alignment graphs. However, there are also profitable alignment graphs with mismatching nodes. Moreover, all the special nodes discussed in Section IV-C also contribute to our final result.

B. Code-Size Reduction on Full Programs

Table I shows the code size reductions achieved by RoLAG on full programs from the MiBench and SPEC 2017 benchmark suites. LLVM's loop rerolling technique is never successfully triggered on these programs. Since we have loop unrolling disabled and these programs have very few loops partially unrolled by hand, LLVM is very ineffective.

RoLAG, on the other hand, is able to find several optimization opportunities. Due to the size of these full programs, RoLAG's impact on code size reduction becomes noticeable when several loops are successfully rolled in the same program. For the largest program, `526.blender_r`, RoLAG performs 2580 successful loop rolling operations, reducing the final binary by almost 88 KB. However, the highest percentage reduction was around 2.7%, achieved on the `511.povray_r` program, where 480 loop rolling operations were successfully performed.

C. Testing the Limits with the TSVC Suite

Our previous experiments have shown that the LLVM's loop rerolling technique is ineffective in most real-world code. In this section, we provide a detailed analysis of both techniques on small kernels from the TSVC suite containing partially unrolled loops. The ideal scenario for LLVM's rerolling technique is represented by such partially unrolled loops that can be perfectly rerolled. Moreover, the TSVC suite provides a variety of code patterns that we use to expose limitations on both techniques. For our evaluation on TSVC, we have forced all its inner loops to unroll by a factor of 8.

Figure 17 shows the reduction, in *text* size, obtained by each technique. In this figure, we are only showing the benchmarks affected by either one of the techniques, except for the average which is computed across all 151 kernels in TSVC. Even though this setup favors LLVM's loop rerolling technique, since it was designed specifically for partially unrolled loops, it was limited to only 38 kernels while RoLAG was able to profitably reroll 84 loops. LLVM achieves an average reduction of 13.69% across all kernels and RoLAG an average of 23.4%. For partially unrolled loops that both techniques are able to reroll, LLVM tends to have a slightly better result as it reuses the same loop for rerolling while RoLAG currently creates a new inner loop. This could be improved by either running a loop flattening pass [31], [32], [33] after RoLAG or simply making it loop aware. However, it is important to highlight that RoLAG is applied to all basic blocks in the program, while LLVM is limited to partially unrolled loops.

LLVM's loop rerolling is only able to handle loops performing simple array operations, such as array initialization and element-wise addition, loops with reduction trees, and some loops with indirect memory access. Unlike vectorization [23], loop rolling can still be applied in the presence of indirect memory accesses since it is guaranteed to preserve the order of all memory accesses, as discussed in Section IV-D. However, it is well-known that loop unrolling tends to enable other optimizations [34], [23], such as common sub-expression elimination, limiting LLVM's ability to reroll the loop.

For the TSVC suite, we can also compare our result with an oracle since we have the original version of the source code prior to the partial unrolling. Figure 18 shows this comparison for all the kernels, sorted by the RoLAG's numbers, similar to Figure 17. The oracle shows that there is still room for improvement, even though RoLAG represents a significant improvement over prior techniques. By examining each one of these kernels, we can identify which features contribute the most for RoLAG and which ones are still missing.

Figure 19 shows a breakdown of the types of nodes from profitable alignment graphs found in the TSVC suite. This

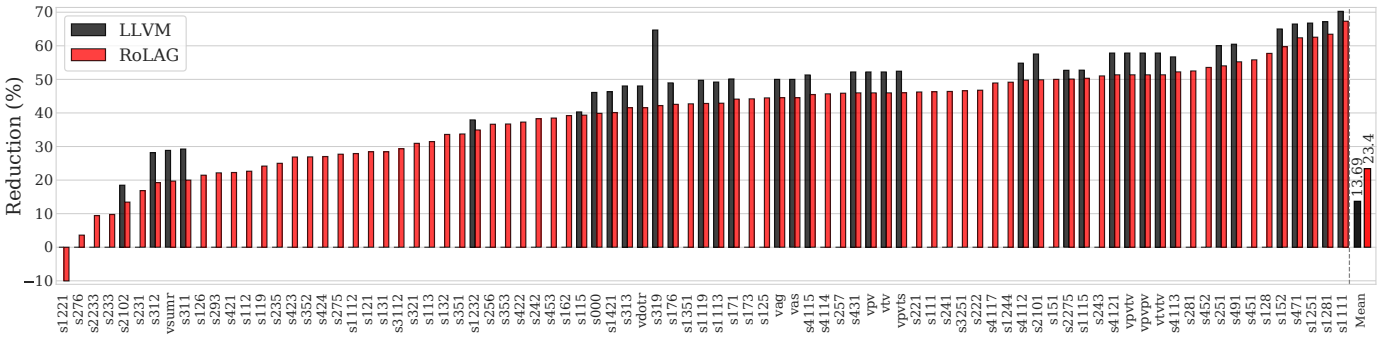


Fig. 17: Code size reduction on the TSVC benchmark suite. RoLAG reduces the final binary by an average of 23.4%.

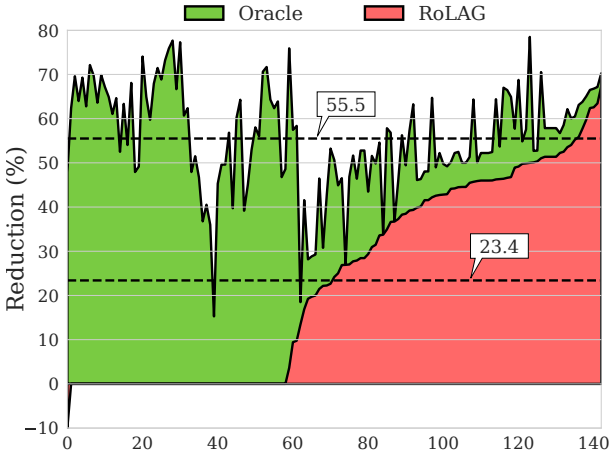


Fig. 18: Comparison of the reduction achieved by an oracle and RoLAG across the whole TSVC suite.

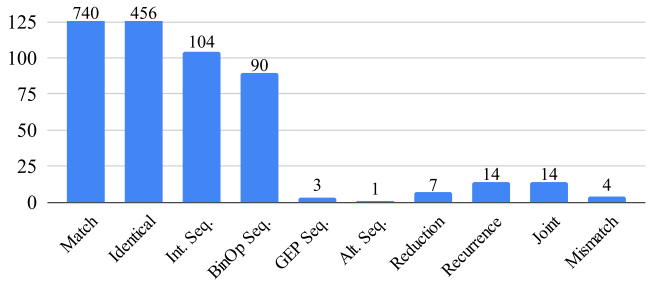


Fig. 19: Breakdown of the different types of node used in profitable alignment graphs across the TSVC suite.

breakdown follows a similar pattern to that from the AnghaBench. If we disable the special nodes, RoLAG can only profitably reroll 19 loops, instead of 84.

There are two profitable alignment graphs with mismatching nodes, namely, the kernels `s452` and `s4117` which contain two nodes each with mismatching constants. Since both kernels have alignment graphs with over 20 nodes, the benefit of rolling them into loops outweigh the cost needed to handle the mismatching nodes. Their sequences of mismatching constants stem from certain transformations on the partially unrolled loop. For example, the compiler may apply strength reduction prior to our loop rolling, causing some addition operations to be transformed into *bitwise-or* operations. If unable to prove

```
for (int i = 0; i < LEN_1D; i++) {
  if (b[i] > 0.f)
    a[i] += b[i] * c[i];
}
```

(a) Kernel `s271`. A loop with multiple basic blocks.

```
for (int i = 0; i < LEN_1D; i++) {
  if ((ABS(a[i])) > max)
    max = ABS(a[i]);
}
```

(b) Kernel `s3113`. A max-based reduction loop.

Fig. 20: Example of two loops that are currently unsupported by both LLVM and RoLAG.

their equivalence, RoLAG uses the special node for binary operations, creating a special node for the addition and another for the *bitwise-or* and filling in the difference with their neutral elements. We plan to improve this as future work.

After examining the loops that both LLVM and RoLAG were unable to reroll, we can identify the main missing features. The most prominent of them are the 26 loops with multiple basic blocks. Figure 20a shows one such example. Because both techniques are limited to a single block, all these opportunities were missed. The remaining cases can be improved by extending several of the capabilities already provided by RoLAG. For example, *min/max* reduction trees are currently unsupported, as shown in Figure 20b, but because this conditional construction is lowered to a *select* instruction, the single block solution should suffice for this example.

D. Performance Overhead on TSVC

RoLAG results in an average slowdown of 0.8 across TSVC. This performance impact is expected due to the nature of the benchmark suite and the loop rolling transformation. Since the TSVC suite was designed with a focus on instruction level parallelism, it benefits heavily from loop unrolling (as well as vectorization). Ideally, the compiler would have profiling information when optimizing for performance where it could disable RoLAG on *hot* basic blocks. However, the focus of this paper is code size optimization, where we have shown the effectiveness of RoLAG.

VI. Related Work

Compiler optimizations for code-size reduction have existed since the very beginning of optimizing compilers [5].

These include redundancy elimination [6], [7], local code motion [35], constant propagation [8], *dead* and unreachable code elimination [9].

A. Lessons Learned from the SLP Vectorizer

Our technique was inspired by the Superword Level Parallelism (SLP), which is a vectorizer for straight-line code [13], [14]. The SLP also starts by collecting seed instructions that are likely to lead to vectorization. Afterwards, an SLP graph is built from the seeds by following their use-def chains. Finally, vectorized code is generated from the SLP graph, if deemed profitable. Several of the optimizations devised to uncover more isomorphism for the SLP vectorizer has the potential to be adapted to our loop rolling technique [36], [25], [37]. For example, the Look-Ahead SLP [36] proposes an improved operand reordering heuristic based on knowledge collected by scanning the code beyond the commutative operations being currently considered for vectorization.

Even though SLP vectorization can potentially reduce code size, it would be restricted to target processors with vector units. Moreover, loop rolling has fewer restrictions than such vectorizers, as the rolled loop is still executed sequentially and the loop count is not limited by the size of the vector unit.

B. Loop Rerolling

Existing work on loop rerolling focus on code replication resulting from loop unrolling, allowing it to be fully rerolled.

Early proposals of loop rerolling had an integrated approach with loop unrolling, where the compiler would fully unroll a loop, analyze or optimize the unrolled loop, and reroll the optimized code back to a loop [38], [39]. In these integrated approaches, the compiler has full information of the prior unrolling while performing the loop rerolling. However, these techniques are limited to loops with known trip count and are target-dependent as they are performed at the binary-level.

Stitt and Vahid [40] propose a loop rerolling technique that work at binary-level by identifying contiguous sequences of repeating code that can be transformed into a loop. All the replicas must have exactly the same instructions appearing in the same order, with corresponding instructions operating over the same registers. Because they work at the binary-level, these solutions are target dependent.

These target-dependent loop rerolling techniques have fallen in priority and are not found in production compilers. Instead, they favor techniques adapted to work directly on their intermediate representation, such as that discussed in Section II.

VII. Conclusion

In this paper, we have proposed a novel loop rolling technique for straight-line code based on a bottom-up graph alignment solution. We show how our alignment strategy can identify and represent several patterns of code in the rolled loop. It includes a powerful scheduling analysis that enables our loop rolling to go beyond contiguous sequences of repeating code, allowing instructions to be rearranged. Finally, we demonstrate the effectiveness of our novel technique on real-world code, where it is triggered several orders of magnitude more often than LLVM's rerolling, resulting in meaningful reductions on real-world functions.

As discussed in Section V-C, our technique currently works only within a single basic block, however, future work could extend it to find isomorphic code across different blocks from the same function. We could also extend this further to incorporate control flow in the graph alignment, allowing us to roll loops containing multiple blocks. Optimizations developed for the SLP vectorizer can also be adapted to our loop rolling technique. We also plan to investigate the impact of loop rolling on other optimizations such as loop vectorization and register allocation, as well as its impact on the instruction cache. Finally, we plan to explore the use profiling information to reduce the impact of loop rolling on hot basic blocks.

Acknowledgment

This work has been supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/L01503X/1 (CDT in Pervasive Parallelism). This work was also supported by the Royal Academy of Engineering under the Research Fellowship scheme. Finally, this work was also supported by a UK RISE Grant and a Huawei Research Grant.

Appendix

A. Abstract

This artifact provides the LLVM source code that implements RoLAG, the contribution of this paper, as well as the LLVM loop rerolling technique. Both optimizations are implemented on top of LLVM-12.0.0. Also included are the scripts that install the necessary dependencies, build LLVM, run the experiments, and reproduce the main figures in this paper. We include the TSVC benchmark suite and a subset of the AnghaBench. The artifact will build the benchmark suite under three strategies: no loop (re)rolling, LLVM's loop rerolling, and RoLAG. It will then generate plots comparing their code size reduction. The scripts can be easily extended to handle more benchmarks. The artifact has a minimal set of requirements that should be already met in most development systems running a modern version of Linux. We provide installation scripts to handle any missing dependencies automatically.

B. Artifact check-list (meta-information)

- **Algorithm:** Loop rolling optimization
- **Program:** TSVC and AnghaBench benchmark suites.
- **Compilation:** GCC-5.1 or above, Clang 3.5 or above.
- **Run-time environment:** Any relatively recent Linux system. Minimal dependencies should be already met. Otherwise *sudo* might be required. If not present, Python3 and cmake will be installed locally.
- **Hardware:** Any x86-64 machine.
- **Metrics:** Object file size.
- **Output:** A set of plots (.pdf). Processed data in CSVs.
- **Experiments:** One bash script to setup the environment, another one to run the experiments and produce all plots.
- **How much disk space required (approximately)?:** 15 GB.

- **How much time is needed to prepare workflow (approximately)?:** ˜30-60 minutes.
- **How much time is needed to complete experiments (approximately)?:** Less than 30 minutes.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License 2.0
- **Archived (provide DOI)?:** 10.6084/m9.figshare.17263274

C. Description

1) How delivered: The artifact is publicly available. It can be downloaded as a `tar.xz` archive from `https://figshare.com/ndownloader/files/31903286`. We provide bash scripts that automate the installation and use of this artifact.

2) Dependencies: A modern Linux installation is assumed. Most of the hard prerequisites below will be already installed on most development systems. If not, you will have to install them using the package manager, either manually or through `install-requirements.sh`.

- GCC-5.1+ or Clang-3.5+
- 32-bit libgcc (gcc-multilib)
- GNU Make 3.79+
- CMake 3.13.4+
- wget and git
- Python3 3.6+ and pip3
- numpy, scipy, and matplotlib

3) Data sets:

- TSVC benchmark suite

We include the source code for the TSVC benchmark suite. We provide two versions of its source code: the first with all its main loops fully rolled; the second with all its main loops partially unrolled. The fully rolled version acts as the oracle for our evaluation.

D. Installation

Download our artifact from the archive and `untar` it:

```
$ wget https://figshare.com/ndownloader/files/31903286 -O cgo22-rolag-artifact.tar.xz
$ tar -xf cgo22-rolag-artifact.tar.xz
$ cd cgo22-rolag-artifact
```

If any of the the dependencies above is missing, run the following command line:

```
$ bash install-requirements.sh  # This requires sudo access
```

In order to download and build our LLVM version, which includes the proposed loop rolling technique, run the following command line:

```
$ bash setup.sh # This will take about half an hour
```

This command will `git clone` our online repository and run the build script.

E. Experiment workflow

The artifact reproduces the results in Sections V.A and V.C, namely, figures 15, 17, and 18. The general workflow of each experiment is the following:

To execute the TSVC experiment run:

```
$ bash run-tsvc.sh  # This will take less than a minute
```

To execute the AnghaBench experiment run:

```
$ bash run-angha.sh  # This will take less than half an hour
```

This command will automatically run the experiments and produce the raw CSV numbers and the final plots in PDF.

F. Evaluation and expected result

After running the experiment, the final plots will be in folder `results`. The scripts will generate the following figures in the paper:

- Fig. 15: `results/angha-curve-code-reduction.pdf`
- Fig. 17: `results/tsvc-bars-code-reduction.pdf`
- Fig. 18: `results/tsvc-curve-code-reduction.pdf`

These figures should be similar to the ones in the paper but not necessarily identical. Object file sizes may slightly vary depending on the machine settings and other aspects that may affect the decision making of the optimizations and code generation.

REFERENCES

[1] T. J. Edler von Koch, I. Böhm, and B. Franke, "Integrated instruction selection and register allocation for compact code generation exploiting freeform mixing of 16- and 32-bit instructions," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 180–189.

[2] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press, 2019, pp. 149–163.

[3] M. Chabbi, J. Lin, and R. Barik, "An experience with code-size optimization for production iOS mobile applications," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. US: IEEE Press, 2021, pp. 1–12.

[4] T. Damásio, V. Pacheco, F. Goes, F. Pereira, and R. Rocha, "Inlining for code size reduction," in *25th Brazilian Symposium on Programming Languages*, ser. SBLP'21. New York, NY, USA: Association for Computing Machinery, 2021, p. 17–24.

[5] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, p. 378–415, mar 2000.

[6] J. Cocke, "Global common subexpression elimination," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: Association for Computing Machinery, 1970, p. 20–24.

[7] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow, "Partial redundancy elimination in SSA form," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 3, p. 627–676, May 1999.

[8] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, p. 181–210, Apr. 1991.

[9] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[10] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: ACM, 2014, pp. 85–94.

[11] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Effective function merging in the SSA form," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 854–868.

[12] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, K. Hazelwood, and H. Leather, "HyFM: Function merging for free," in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 110–121.

[13] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 145–156.

[14] I. Rosen, D. Nuzman, and A. Zaks, "Loop-aware SLP in GCC," in *GCC Developers' Summit*, 2007.

[15] L. developers. (2021) Looprerollpass.cpp. [Online]. Available: https://github.com/llvm/llvm-project/blob/7217b01481566092c440f249d65a10efee8c50ad/llvm/lib/Transforms/Scalar/LoopRerollPass.cpp

[16] M. Wolfe, "Beyond induction variables," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 162–174.

[17] L. developers. (2021) aegis128_save_state_neon. [Online]. Available: https://github.com/torvalds/linux/blob/master/crypto/aegis128-neon-inner.c\#L41

[18] ——. (2021) hdmi_wp_audio_config_format. [Online]. Available: https://github.com/torvalds/linux/blob/master/drivers/video/fbdev/omap2/omapfb/dss/hdmi_wp.c\#L203

[19] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[20] K. Matsuzaki, Z. Hu, and M. Takeichi, "Parallelization with tree skeletons," in *Euro-Par 2003 Parallel Processing*, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 789–798.

[21] P. Ginsbach and M. F. P. O'Boyle, "Discovery and exploitation of general reductions: A constraint based approach," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 269–280.

[22] R. C. Rocha, L. F. Góes, and F. M. Pereira, "Automatic parallelization of recursive functions with rewriting rules," *Science of Computer Programming*, vol. 173, pp. 128–152, 2019, brazilian Symposium on Programming Languages (SBLP). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642318300066

[23] R. C. O. Rocha, V. Porpodas, P. Petoumenos, L. F. W. Góes, Z. Wang, M. Cole, and H. Leather, "Vectorization-aware loop unrolling with seed forwarding," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13.

[24] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 1–11.

[25] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes, "VW-SLP: Auto-vectorization with adaptive vector width," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018.

[26] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. Ferreira Guimarães, and F. M. Quinão Pereira, "Anghabench: A suite

[32] A. M. Ghuloum and A. L. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 58–67.

with one million compilable C benchmarks for code-size reduction," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 378–390.

[27] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.

[28] SPEC, "Standard Performance Evaluation Corp Benchmarks," http://www.spec.org, 2021.

[29] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: A test suite and results," in *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '88. Washington, DC, USA: IEEE Computer Society Press, 1988, p. 98–105.

[30] L. developers. (2021) copy_vmcs12_to_enlightened. [Online]. Available: https://github.com/torvalds/linux/blob/master/arch/x86/kvm/vmx/nested.c\#L1814

[31] R. v. Hanxleden and K. Kennedy, "Relaxing simd control flow constraints using loop transformations," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 188–199.

[33] S. Pop, R. Yazdani, and Q. Neill, "Improving GCC's auto-vectorization with if-conversion and loop flattening for amd's bulldozer processors," in *GCC Developers' Summit*. Citeseer, 2010, p. 89.

[34] J. W. Davidson and S. Jinturkar, "Aggressive loop unrolling in a retargetable, optimizing compiler," in *Compiler Construction*, T. Gyimóthy, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 59–73.

[35] S. Heilala, "Optimizing java for size: Compiler techniques for code compaction," 2007.

[36] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes, "Look-Ahead SLP: Auto-vectorization in the presence of commutative operations," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 163–174.

[37] V. Porpodas, R. C. O. Rocha, E. Brevnov, L. F. W. Góes, and T. Mattson, "Super-Node SLP: Optimized vectorization for code sequences containing operators and their inverse elements," in *Proceedings of the 2019 International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press, 2019, pp. 206–216.

[38] B. Su, S. Ding, and L. Jin, "An improvement of trace scheduling for global microcode compaction," in *Proceedings of the 17th Annual Workshop on Microprogramming*, ser. MICRO 17. IEEE Press, 1984, p. 78–85.

[39] B. Su, S. Ding, and J. Xia, "Urpr—an extension of urcr for software pipelining," in *Proceedings of the 19th Annual Workshop on Microprogramming*, ser. MICRO 19. New York, NY, USA: Association for Computing Machinery, 1986, p. 94–103.

[40] G. Stitt and F. Vahid, "New decompilation techniques for binary-level co-processor generation," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, 2005, pp. 547–554.