Evaluating the Effect of Loop Rolling on Code Size Reduction

Xuan Peng
Information Networking Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: xuanpeng@andrew.cmu.edu

Zheng-Lun Wu
Information Networking Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: zhengluw@andrew.cmu.edu

1. Introduction

1.1. The Problem/Opportunity

Code size is important especially in resource constrained environments, thus a lot of compiler optimizations are related to it. There are already a lot of optimizations done to reduce code size, some in instruction level while others in basic block level. For example, common sub-expression elimination, partial redundancy elimination, constant propagation, and dead and unreachable code elimination are all common ways to remove redundant code and further reduce code size. There are other ways like function-merging - merge functions that are identical into a single one, or function-outlining - putting equivalent basic blocks into a single function that are other ways to reduce code size.

Another way to reduce code size is loop rolling, it is a method performed in the instruction level. Although some previous work are already done, we found that there are still plenty opportunities to reduce code in this way that haven't being fully explored yet.

1.2. Your Approach

We are using loop rolling to reduce code, which means by identifying straight-line code in the same block that shares a certain pattern, and then transform it into a loop.

Our approach only works in a basic block in instruction level. It will first iterate the instructions in the basic block, and collect "seed instructions". Seed instructions are store instructions or call instructions, store instructions are grouped based on data type and base address while call instructions are grouped based on the address of callee. Seed instructions are the target instructions that we are going to rolled in loops.

For each group of seed instructions, we will build an alignment graph out of it. An alignment graph consists of a lot of nodes, which wraps a group of instructions. We traverse the use-def chains of the instructions in the nodes and stops when a node contains all constant values or function parameters. A node is marked "matching" if all the instructions are "identical", ((1) having the same opcode, (2) same type, (3) corresponding operands has equivalent type), or all the values are the same. Otherwise, the node is marked "mismatching".

To reduce the number of mismatching nodes, we can turn it to "special" nodes by identifying certain code patterns and then transform them, so we are able to roll them into loops. For example, if a node contains an arithmetic sequence and the common difference of successive members is 3, we will mark it as a mismatching nodes initially since every value in the node is different. But, in fact, by transforming it to something related to the induction variable 3 * i, we turn it into a special node. All values in this node can use 3 * i to represent it, and it is possible that we can roll this into a loop in later steps.

After the alignment graph is built, we do an analysis to see if it is possible for us to move the related code around, if it is valid to move the instructions in the alignment graph around, we move to the code generation step. For each alignment graph, we will generate try to move the code into a newly created loop block. Finally, we do a profitability analysis by comparing the code size of the loop and the original basic block to see if rolling the loop generates smaller code size. If it fails to generate smaller code, we roll back to the original version.

1.3. Related Work

For other kinds of strategies, there are already lots of LLVM transform passes and research done to reduce the code size.

Regarding to the loop rolling method, there is also one that is implemented in a LLVM pass.

It is able to turn code from this

```
void bar(int *x) {
  for (int i = 0; i < 500; i += 3) {
    foo(i);
    foo(i+1);
    foo(i+2);
  }
}
```

into this

```
void bar(int *x) {
```

```
  for (int i = 0; i < 500; ++i)
    foo(i);
}
```

Although this method do help identify a lot of opportunities to reduce code size, it is way too conservative since it only handles partially unrolled loops that can be further rolled into a loop. It cannot turn straight line code into loops.

```
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.

Figure 1. Example of loop rolling opportunity that cannot be identified by LLVM pass

```
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.

Figure 2. Real-world code from the Linux repository that should be rolled into a loop

Figure 1 and Figure 2 are both examples that are easy for human to identify opportunities to roll into loop and reduce code size, but unfortunately are not handled by the LLVM LoopRerolling pass.

1.4. Contributions

The contribution of our work is that, by identifying multiple code patterns, we are able to roll straight-line code into loops and reduce code size. This hasn't being done before and can effectively do code size compression.

2. Loop Rolling

We will present the steps to do loop rolling below. The psuedocode of the whole process is in Figure 3.

```
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)
```

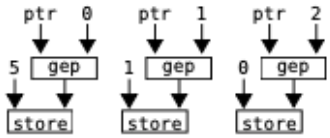
Figure 3. Pseudocode for the whole loop rolling process

2.1. Collect Seed Instructions

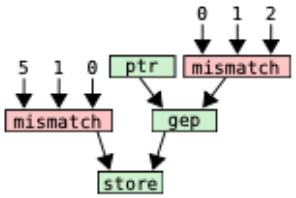
For starters, we will scan every instruction in a basic block and collect groups of seed instructions.

We define seed instructions as store instructions and call instructions. These instructions are grouped based on similar aspects. For store instructions, they are grouped by data type and base address; while for call instructions, they are grouped by the callee.

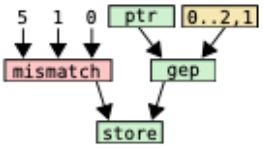
Figure 4.a is an example of a group of seed instructions, each SSA graph represents an instruction. The SSA graph here is a representation of the use-def chain of the instruction.



(a) Input SSA graphs.



(b) Simple alignment graph.



(c) Improved alignment graph.

Figure 4. The alignment graph built after the first three steps

2.2. Build Alignment Graph

After we got groups of seed instructions, we will start building alignment graphs by traversing the use-def chains of instructions in the group. The alignment graph is named from the fact that it is built from aligning isomorphic codes, it uses nodes to represent a group of instructions.

At this step, there are two types of nodes: (1)Matching nodes: all the instructions/values in the node are identical to each other and (2) Mismatching nodes: nodes that are not matching nodes. Here, the equivalence of instruction is defined by having (1) same opcode, (2) same type, (3) corresponding operands with same type.

After we built each node, we will traverse the use-def chain by building nodes that is formed by the operands of the instructions in the current node. Figure 5 is a pseudocode of the process of building alignment graphs and Figure 4.b is an example of the alignment graph built from Figure 4.a.

This step will stop when all the instructions/values in a node are (1) constant values and (2) function parameters.

```
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
```

Figure 5. The pseudocode of the process of building alignment graph

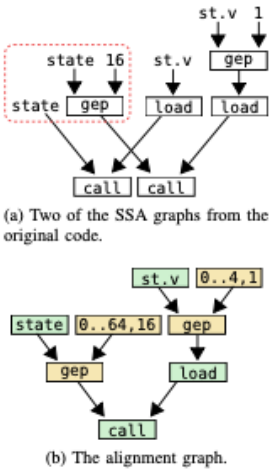
2.3. Apply Special Code Pattern

After the previous step, there are still a lot of mismatching nodes, making it hard to do loop rolling. We can identify and apply certain code patterns to mismatching nodes to change them into matching nodes. here we will label them as "special" nodes to differentiate them from nodes that are matching from the beginning.

2.3.1. Monotonic Integer Sequences. This kind of code pattern is to identify whether an arithmetic sequence exists in all the values in the node.

Figure 4.c is an example of how we can apply this code pattern to matching nodes to Figure 4.b, so that we can find out opportunity to roll loops.

2.3.2. Neutral Pointer. Figure 6 is an example of the neutral pointer pattern.



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

(b) The alignment graph.

Figure 6. Example of neutral pointer pattern

This pattern often arises when performing pointer operations, a common example is the `getelementptr` instruction (shortened as `gep`), which 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.

2.4. Scheduling Analysis

Before doing code generation, we need to first make sure that switching the code sequence in the basic block will not alter program semantics.

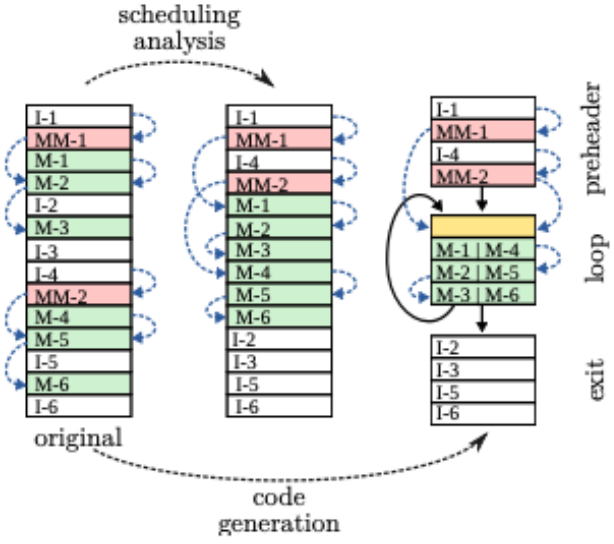


Figure 7. Scheduling Analysis

Figure 7 shows how we are going to rearrange instructions to perform loop rolling. If any pair of instructions that

cannot switch their execution order, i.e. one of them has side effects, then the scheduling analysis failed and we cannot perform loop rolling.

The rearrangement needs to make sure that (1) Mismatching nodes and their dependences precedes instructions in matching nodes (2) Instructions outside of the alignment graph and is not depended by instructions in the alignment graph are placed after instructions in matching nodes (3) Instructions outside of the alignment graph but instructions in the alignment graph depends on, are placed before instructions in matching nodes.

2.5. Generate Code

If we pass the test in previous steps, we can start generate the code we need. This is done for each alignment graph.

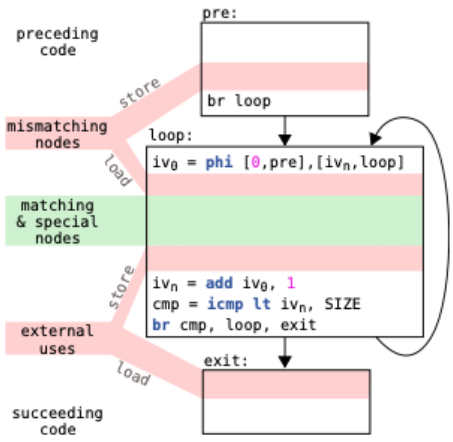


Fig. 14. Diagram of the code generation.

Figure 8. Code Generation

The first step is to generate basic blocks and insert control instructions in the basic blocks which are needed for creating a loop. The three blocks needed are: (1) preheader: contains instructions needed for handling mismatching nodes, (2) loop: contains control instructions of the loop and also instructions from matching/special nodes and (3) exit. Figure 8 shows the basic block needed for code generation.

For the next step, we construct the def-chain of the instructions by doing post-order traversal of the alignment graph. If this node is a matching node, we copy the instruction into the loop. If this node is a mismatching node, we create a stack array in preheader and add store instructions in the preheader to store all the values in this array. Then in the loop basic block, we add a single load instruction to load from the stack array that we just created in the preheader. If this node is a special node, we also copied it into the loop, but the instructions we copied may need to do some transformations related to the induction variable.

2.6. Profitability Analysis

After the code generation step, it is possible that we may end up in a larger code size cause although we are rolling into loops, we may still add instructions like for mismatching nodes in the preheader basic block. It is thus necessary to compare the code size of original basic block and the newly generated loop. If the code size turns out to be even larger, we will abandon loop rolling of this certain seed group and stick to the original code.

For calculation of the code size, this is done by LLVM's built-in cost model, target transformation interface(TTI). We sum up the cost of every instruction in the original basic block and compare it with the instructions in the newly generated loop.

3. Experimental Setup

The experimental environment of the original paper targets 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.

However, due to the fact that we have no access to machines other than our laptops, we perform our experiments on a 8-core M1 chip with 16 GiB of RAM, running macOS BigSur 11.4.

4. Experimental Evaluation

Show the results of your experiments, and provide detailed analysis of these results. This section might start out with the overall effectiveness of your technique, followed by various subsections that examine specific aspects of your design in more detail (e.g., sensitivity analysis to various design parameters).

We split the evaluation section into 2 parts as unit testing evaluation and benchmark testing evaluation.

4.1. Unit Testing Evaluation

This part of work shoulders 2 major responsibilities. Firstly, it is used to verify the correctness of our implementation. Secondly, it portraits the most common cases where our loop rolling technique could achieve desired impact. Specifically, we designed and conducted separate unit tests for 2 special code patterns. In the following paragraphs we will walk through the test cases in detail.

4.1.1. Monotonic Integer Sequence. As discussed in previous paragraphs, a monotonic integer sequence is such a sequence that the differences between any 2 consecutive elements are equal. You would possibly note an interesting case, where all the elements are the same. And that actually makes the node in the alignment graph identical. So this is pretty trivial, and interestingly this identical case is exactly one of the only few scenarios that the LLVM's

```
define dso_local void @_Z4monoPi(i32* %x) #0 {
entry:
  %arrayidx = getelementptr inbounds i32, i32* %x, i32 0
  store i32 3, i32* %arrayidx, align 4
  %arrayidx1 = getelementptr inbounds i32, i32* %x, i32 1
  store i32 6, i32* %arrayidx1, align 4
  %arrayidx2 = getelementptr inbounds i32, i32* %x, i32 2
  store i32 9, i32* %arrayidx2, align 4
  %arrayidx3 = getelementptr inbounds i32, i32* %x, i32 3
  store i32 12, i32* %arrayidx3, align 4
  %arrayidx4 = getelementptr inbounds i32, i32* %x, i32 4
  store i32 15, i32* %arrayidx4, align 4
  %arrayidx5 = getelementptr inbounds i32, i32* %x, i32 5
  store i32 18, i32* %arrayidx5, align 4
  ret void
}
```

Figure 9. IR of code example1

```
define dso_local void @_Z4monoPi(i32* %x) #0 {
entry:
  br label %preheader

preheader:
  br label %loop

loop:
  %0 = phi i32 [ 0, %preheader ], [ %4, %loop ]
  %1 = mul i32 %0, 1
  %2 = getelementptr inbounds i32, i32* %x, i32 %1
  %3 = mul i32 %0, 1
  store i32 %3, i32* %2, align 4
  %4 = add i32 %0, 1
  %5 = icmp ult i32 %4, 6
  br i1 %5, label %loop, label %6

6:
  ret void
}
```

Figure 10. IR of code example1 after loop rolling

loop rerolling pass could be applied. We care more about the cases where the difference is non-zero. e.g., a sequence like 3, 6, 9, 12.

For Monotonic Integer Sequence, the test case we designed is as shown below as example1.

```
void mono(int *x) {
    x[0] = 3;
    x[1] = 6;
    x[2] = 9;
    x[3] = 12;
    x[4] = 15;
    x[5] = 18;
}
```

And we provide its representation of LLVM's IR form in fig9. In fig10, we present the IR of the code after the loop rolling transformation pass. As is shown, after the loop rolling transformation, the generated IR is totally different from the original one, in that it includes 3 more basic blocks which represent the preheader, loop body, and loop exit respectively. With the help of this test case, the correctness of our implementation of Monotonic Integer Sequence rolling is demonstrated.

We then further take a step forward and see the effect to the code in a quantitative way. LLVM provides such an interface as *TargetTransformInfo.getInstructionCost()* that can return the estimated cost of an instruction under targeted platform. It takes 2 arguments. The first one is the pointer to the instruction that we want to gauge. The second one is an enum value specifying what kind of cost we want it to gauge, including Reciprocal throughput, the latency of instruction, instruction code size, and the weighted sum of size and latency. Here we care only the instruction code size. The interface implementation will collection the information of current machine platform transparently, and finally return the estimated code size cost for the instruction.

We calculated the overall code size of the program by accumulating the cost of each individual instruction. And we do this both before and after the transformation. As a result, we can compare the 2 versions of costs and decide the next step we are going to choose. Below is the shorter example:

```
void mono(int *x) {
    x[0] = 3;
    x[1] = 6;
    x[2] = 9;
}
```

And below is the longer example:

```
void mono(int *x) {
    x[0] = 3;
    x[1] = 6;
    x[2] = 9;
    x[3] = 12;
    x[4] = 15;
    x[5] = 18;
    x[6] = 21;
    x[7] = 24;
    x[8] = 27;
    x[9] = 30;
    x[10] = 33;
    x[11] = 36;
    x[12] = 39;
}
```

Figure 11 shows the program output on console for the shorter straight line code example. And figure 12 shows the longer one's.

```
Old cost: 5
New cost: 9
user@user-VirtualBox:/media/sf_CMU15745-Project/pass$
```

Figure 11. console output of shorter example

```
Old cost: 25
New cost: 9
user@user-VirtualBox:/media/sf_CMU15745-Project/pass$
```

Figure 12. console output of longer example

Just as what we mentioned before, not all loop rolling transformation is profitable.The transformation for the shorter code is a good example: the original code size is 5, while the size of the transformed code is 9. But for the longer case, the transformed code got a overall code size of 9. Compared with the original size of 25, the transformation reduces the size of the program by 16/25 = 64%.

You may note the sizes of both programs after the transformation interesting: though their initial size differs a lot, their code sizes after transformation are all 9. The rationale behind is simple: those 2 programs are actually rolled into a similar loop. The preheader, loop exit, and most of the loop body is the identical. The only difference is the number of iterations, which is reflected near the bottom of loop body by branch instructions.

4.1.2. Neutral Pointer Operations. Technically speaking, the pattern of Neutral Pointer Operations is very similar with the one of Monotonic Integer Sequence. What we do in this pattern is to try to identify if:

- for a mismatched node of the alignment graph, there is only one non-instruction value. The rest values are all pointer arithmetic instructions.
- the non-instruction value and the base addresses of the pointer arithmetic instructions should be the same.
- if we modify the only non-instruction value also to a pointer arithmetic instruction by adding a offset of 0 or multiply by 1, the value sequence would show special feature. e.g., arithmetic progression and geometric progression.

The example we used for neutral pointer operations is as below:

```
void foo(int* state, int** x){
    x[0] = state;
    x[1] = state + 16;
    x[2] = state + 32;
    x[3] = state + 48;
    x[4] = state + 64;
}
```

```
define dso_local void @_Z3fooPiPS_(i32* %state, i32** %x) #0 {
entry:
  %arrayidx = getelementptr inbounds i32*, i32** %x, i32 0
  store i32* %state, i32** %arrayidx, align 4
  %add.ptr = getelementptr inbounds i32, i32* %state, i32 16
  %arrayidx1 = getelementptr inbounds i32*, i32** %x, i32 1
  store i32* %add.ptr, i32** %arrayidx1, align 4
  %add.ptr2 = getelementptr inbounds i32, i32* %state, i32 32
  %arrayidx3 = getelementptr inbounds i32*, i32** %x, i32 2
  store i32* %add.ptr2, i32** %arrayidx3, align 4
  %add.ptr4 = getelementptr inbounds i32, i32* %state, i32 48
  %arrayidx5 = getelementptr inbounds i32*, i32** %x, i32 3
  store i32* %add.ptr4, i32** %arrayidx5, align 4
  %add.ptr6 = getelementptr inbounds i32, i32* %state, i32 64
  %arrayidx7 = getelementptr inbounds i32*, i32** %x, i32 4
  store i32* %add.ptr6, i32** %arrayidx7, align 4
  ret void
}
```

Figure 13. IR of example for neutral pointer operations

and the IR is shown in: 13. We can see it's look pretty much just like the example for monotonic integer sequence, except that there is an outlier instruction. So the approach we take here is to transform the pattern from neutral pointer operation to monotonic integer sequence by modifying the only non-instruction value.

4.2. Benchmark Testing Evaluation

For this part, our aim is to find out the actual level of impact that our loop rolling technique could bring to real-world code in terms of the reduction of code sizes. In the original paper [1] of RoLAG, the author claimed that:

- For individual functions, it could achieve an average reduction rate of 9.12% on AnghaBench suite.
- For full program, it could achieve a highest percentage of 2.7% for certain repository in terms of the reduction in overall code size.

However, the way we implement the loop rolling is based on the function pass of LLVM, where we further do transformation upon each individual basic block. Combined with the tight time schedule, we chose to ignore the test on TSVC suite since it is used for full program evaluation.

As an alternative, we wanted to carry out our benchmark test on AnghaBench. But we encountered unexpected issues when using it. Getting it worked takes much more effort than we expected. Since the tests in AnghaBench is directly extracted from real-world well-known code repository, they usually contain much more complex code than what we could handle. In other words, our loop rolling program will crash when taking the functions in AnghaBench as the input. This is a surprise and a lesson learned to us. We are trying to generalize our loop rolling program in order to be more robust and able to handle various kinds of input.

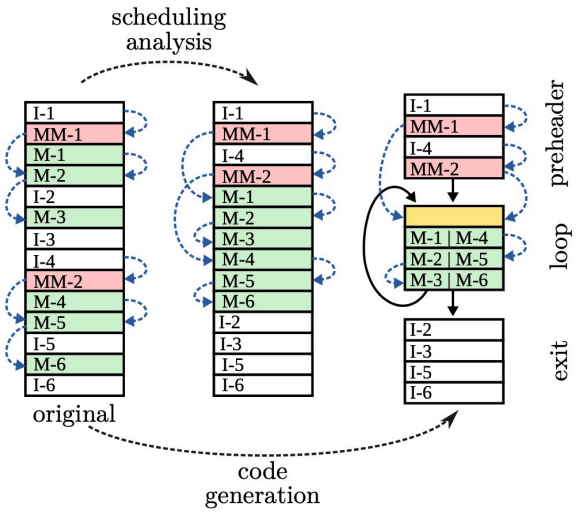


Figure 14. interaction among codegen components

5. Surprises and Lessons Learned

We encountered some surprises along the project. One very important component of the implementation is how to achieve a "rollback-like" mechanism for the transformation. In the cases where the new code size is bigger than the old one's, we will abandon the transformation and go ahead with the original code. However, through the phases of implementation, we may well have made some update to the original code in order for the transformation to continue. e.g., For the pattern of neutral pointer operations, the way we approach it is to transform the problem to the Monotonic Integer Sequence problem. By changing the only non-instruction value to a *getelementptr* instruction with the offset operand of 0, the transformation is finished and we can then go ahead reusing the subsequent processing pipeline of monotonic integer sequence. However, if in the end we find the transformation not profitable, we need to rollback to the original code, while the original code was already modified. The way we can think of to solve the above-mentioned problem is to use a flag bit for every instruction to indicate if this instruction is original or artificial. And for the artificial instructions we also record their original version of instructions so that we can map them back in the rollback phase. But this is not a very graceful design, in that we actually need to loop back to the pipeline component that we've gone through, which destroy the architecture.

For lessons learned technical-wise, first, we realized that the papers does not provide every detail to the reader, so often times we have to experiment different solutions by ourselves. An example is the paper did not mention how to determine the induction variable for a generated loop. Also, the part of code generation is actually one of the most difficult part to implement because there are a lot of details one needs to know before he can make the right move. e.g., the paper uses the figure 14 to illustrate the interaction among the components of alignment graph, scheduling analysis, and code generation. But it does not mention where do we place the generated basic blocks and what are some good practices to keep the generated program robust and correct since it's very easy for readers to ignore some important aspects, including depencencies replacement, uses replacement, etc.

For lessons learned about project management. The most important lesson is it is very easy to under-estimate the workload of a project before one has investigated every component of the project and form a concrete to-do list for each for them. For example, at the planning phase of our project, we thought that we had understood all the patterns that the paper proposed so that we were ready to go. However, we did not expect the amount of work that will be used on exploring and experimenting all kinds of LLVM's API. Also for the AnghaBench, we did not check the code examples inside it carefully before we finished the implementation of the loop rolling program. Later when we found out our program was not able to take most of its tests as input, it was already too late for us to refactor the code.

The second important lesson that we learned is to start early and keep the schedule. At the early stage, both of us did not put much effort into the project as what we planned in the project proposal. Surely one could argue that there are too much workload from various courses taken. But later one will find out that the pressure would only be increasing as the date getting close to the final days, and the only way to conquer a project is to split it into atomic pieces with clear roadmaps of how to solve them in mind, and also adhere to the schedule.

The third lesson is about team collaboration. In the beginning stages of our collaboration, we actually don't know each other very much and we both just keep ourselves isolated in a private zone. For the assignments and the first half of the project, we did not schedule any in-person meetups to discuss. So the ice breaking is pretty slow and the team efficiency is affected. Only later in the project when we started to meet in person then we realize that working face-to-face could bring so much efficiency.

6. Conclusions and Future Work

To conclude, compiler optimizations for code-size reduction have existed since the very beginning of optimizing compilers [2]. These include redundancy elimination [3], [4], local code motion , constant propagation [5], dead and unreachable code elimination [6].

We designed and implemented a generalized code transformation pipeline built upon LLVM as indicated in

the paper [1]. We further digested the main ideas and implemented 2 of the code patterns introduced in the paper. With improvised unit tests, we confirmed the effectiveness of the loop rolling techniques.

For the future work, we will keep iterating the current loop rolling code transformation framework in order to make it more versatile and robust. For example, we are experimenting with the 3rd code pattern of chained dependences, which would further enhance the capability of loop rolling. Also we will rewrite some parts of the code to make it able to take random real-world program as input.

Other important things that can be done but have not been done are:

- adding the capability of processing other kinds of seed instructions. e.g., function call, reduction tree.
- designing a graceful and robust rollback mechanism when profitable analysis shows no improvement.
- processing the other 4 patterns where loop rolling can be applied. This should be built upon the perfection of seed collection.
- experimenting the program on large scale real-world dataset such as AnghaBench.

7. Distribution of Total Credit

Zheng-lun Wu: 55%, Xuan Peng: 45%

References

[1] R. C. Rocha, P. Petoumenos, B. Franke, P. Bhatotia, and M. O'Boyle, "Loop rolling for code size reduction," in *International Symposium on Code Generation and Optimization (CGO) 2022*, 2021.

[2] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Transactions on Programming languages and Systems (TOPLAS)*, vol. 22, no. 2, pp. 378–415, 2000.

[3] J. Cocke, "Global common subexpression elimination," in *Proceedings of a symposium on Compiler optimization*, 1970, pp. 20–24.

[4] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow, "Partial redundancy elimination in ssa form," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 3, pp. 627–676, 1999.

[5] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 2, pp. 181–210, 1991.

[6] K. D. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.