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. Project Description

Code size is important especially in resource-constrained environments. Aside from optimizations like dead-code elimination or common subexpression elimination, there is also another way called loop (re)rolling that we can perform to do code size reduction.

What loop rolling does is to transform equivalent instructions from a basic block to a loop. Although there are existing LLVM passes [**?**] that perform loop rerolling to reduce the code size, it often cannot identify opportunities for code size compression because it cannot handle straight-line code.

The goal of our project is to implement a solution to transform arbitrary pieces of equivalent code into a loop and analyze whether it is profitable (code size reduced successfully) after our transformation.

For evaluation, we will compare our result with the LLVM looprerolling pass [**?**] by benchmarking on test suites like AnghaBench [**?**] and MiBench [**?**].

2. Project URL

We will present the update of this project in this website, including this proposal and further reports.

3. Logistics

3.1. Getting Started

The paper proposed a new way to do loop rolling optimization as opposed to the one provided by LLVM, which is not good enough in that it can only catch one special code pattern and the scope is confined inside loops, which is actually "loop rerolling". It can only refine existing loops, instead of creating new loops. A concrete example for the LLVM's loop rerolling is shown in figure 1.

As we can see, this is far from good enough, since this kind of code pattern is so naive. Few programmers write their code in this way. Therefore, to further improve the effectiveness of loop rolling, more matching code patterns

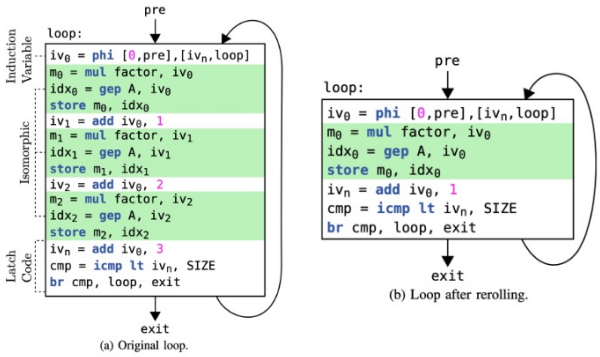


Figure 1. LLVM loop rerolling.

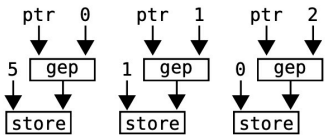
are reasoned and proposed in this paper. We explore several patterns of code that can be leveraged to increase the amount of aligned code in the coming paragraphs with figures for better illustration.

3.1.1. Monotonic Integer Sequences. In figure 2, the original code contains a arithmetic sequence [0,1,2], whose common difference is 1. Therefore we can use the common difference as the loop invariant and represent elements in the sequence with the loop invariant.

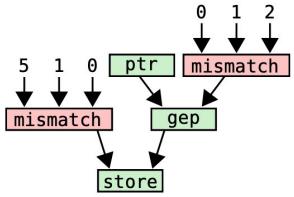
3.1.2. Neutral Pointer Operations. Figure 3 shows the pattern of neutral pointer operation. We can treat it as a special type of monotimic integer sequence, which has a base offset (the pointer variable).

3.1.3. Algebraic Properties of Binary Operations. The paper also exploits algebraic properties of binary operations in order to improve the graph alignment strategy. For example, $a = b$ could be represented with $a = b + 0$, $a = b - 0$, and $a = b * 1$, etc. We try to use the algebraic identities to find more matching blocks.

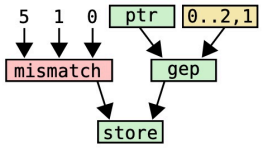
3.1.4. Chained Dependences. Figure 4 shows an example of chained dependency. Basically we want to exploit the idea of phi-node and try to transform the alignment graph of various kinds of blocks with chained dependences (e.g., chained calls) into an iterative form.



(a) Input SSA graphs.



(b) Simple alignment graph.



(c) Improved alignment graph.

Figure 2. Example of a monotonic interger sequence. We catch the sequence [0,1,2] in (b) and convert it from a mismatch to a new form that can be represented with a loop invariant shown in (c)

```
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] );
}
```

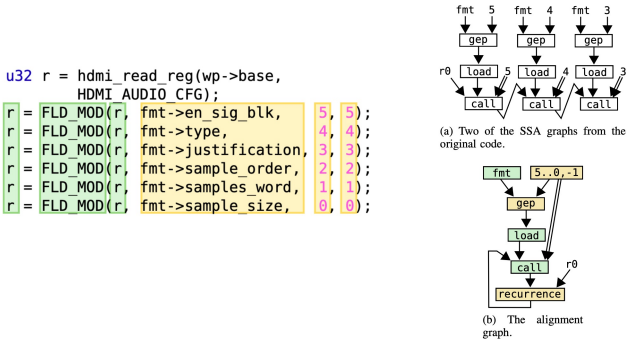
(a) Original function found in the Linux code base.

```
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 3. Example of a neutral pointer operation. It's very similar with monotonic sequence pattern 2, but different in the getelementptr(gep) operation with a given base offset.

3.1.5. Reduction Trees. A reduction tree is a tree where all its internal nodes perform the same associative binary operation. 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, reassociation of floating-point instructions must be explicitly enabled, e.g., via the compilers *fast-math* flag. An example of reduction trees is shown in figure 5.



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

(b) The alignment graph.

Figure 4. Example of a chained dependences pattern.

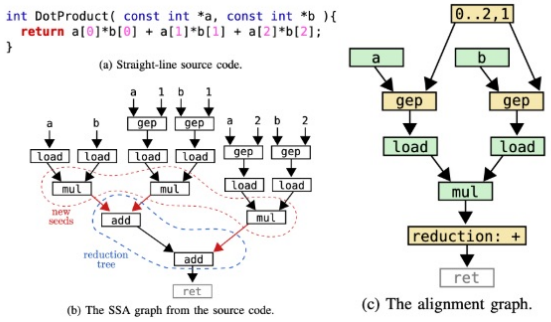
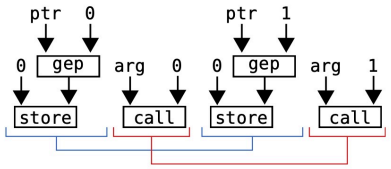
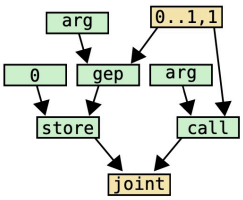


Figure 5. Example of a reduction tree. The whole reduction tree is represented by a special node in the alignment graph.



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



(b) The alignment graph.

Figure 6. Example of two groups of seed instructions combined in a single alignment graph by a joint node.

3.1.6. 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 roll these patterns of code into a single loop. An example of joining alignment graph is shown in figure 6.

3.2. Plan of Attack and Schedule

- **Week 1 (14rd March)**
  Submit project proposal, get familiar with the paper and the materials we need. Run and play with LLVMs loop rerolling pass.
- **Week 2 (21st March)**
  Both work on the initial part: seed collection, alignment graph creation and try to abstract at least 1 code pattern listed in the paper.
- **Week 3 (28th March)**
  - Xuan Peng: Work on code generator to take our alignment graph as input and create the code for rolled loop.
  - Zheng-Lun Wu: Keep working on the other code patterns left.
- **Week 4 (4th April)**
  - Xuan Peng: Keep working on code generation
  - Zheng-Lun Wu: Write the profitability analyzing pass
- **Week 5 (11st April)**
  Prepare and submit a milestone report and discuss with Professor Mowry if there is anything we would like to explore/improve.
- **Week 6 (18th April)**
  Conduct testing on the 4 benchmarks.
- **Week 7 (25th April)**
  Wrap up, finish the project report and prepare for the poster session.

Week 2 to Week 4 would be our critical path since it is the main procedure for creating the loop rolling solution.

3.3. Milestone

By Thursday, April 14th, we plan to finish the implementation and testing of this loop rolling technique. We will prove our progress with a set of selected test cases afterward.

In general, this setting gives us 4 weeks to set up the environment, get familiarized with the algorithm by further reading and reasoning about the paper, and then implement the passes. After that, there are roughly two weeks before the project due date, during which we would conduct the evaluation. i.e., to see how much improvement can we achieve with this technique.

3.4. Resources Needed

The original paper carried out its evaluation under 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. We want to experiment with the same setting. However, so far, we have not found any suitable provided resources. So while we keep searching, we choose to firstly develop the algorithm under our develop environment. Hopefully, we can figure this out in the next 4 weeks before the evaluation phase starts.

3.5. Literature Search

R.C Rocha et al [1]s work is our main guidance for implementing this loop rolling solution, other than that we also look into The source code for llvm loop rerolling pass [**?**]. Although we are going to perform a different way to do loop rerolling, this source code helps us understand the background of loop rerolling which is not clearly explained in the paper.

We will also look into the SLP vectorizer paper [**?**] since the steps we are going to take are somewhat inspired by this paper - collecting seed instructions, tracing use-def chains, generating graphs and analyzing profitability.

Last but not least, several papers [**?**] [**?**] would help us understand more about the basics of loop rolling.

References

[1] L. developers, "Looprerollpass.cpp," https://github.com/llvm/llvm-project/blob/7217b01481566092c440f249d65a10efee8c50ad/llvm/lib/Transforms/Scalar/LoopRerollPass.cpp, 2021.

[2] J. d. S. M. J. R. B. F. G. A.F. daSilva, B.C.Kind and F. Q. Pereira, "A suite with one million compilable c benchmarks for code-size reduction," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)2021*, 2021, pp. 378–390.

[3] D. E. T. M. A. T. M. M. R. Guthaus, J. S. Ringenberg and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*, 2001, pp. 3–14.

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

[5] R. C. O. R. V. Porpodas and L. F. W. G. es, "Vw-slp: Auto- vectorization with adaptive vector width," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018.

[6] S. D. B. Su and L. Jin, "An improvement of trace scheduling for global microcode compaction," in *Proceedings of the 17th Annual Workshop on Microprogramming*, 1984, pp. 78–85.

[7] S. D. B. Su and J. Xia, "Urpran extension of urcr for software pipelining," in *Proceedings of the 19th Annual Workshop on Microprogramming*, 1986, pp. 94–103.