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

Tianao Ge, Zewei Mo, Kan Wu, Xianwei Zhang, Yutong Lu

Sun Yat-sen University

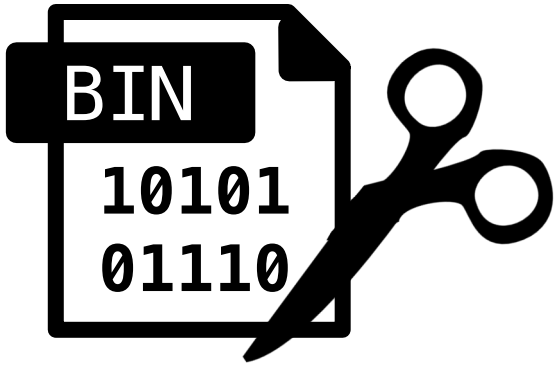
中山大學
SUN YAT-SEN UNIVERSITY

LCTES 2022

Code Size Matters

- Memory is very restrictive on resource-constrained devices
  - embedded and IoT devices
  - desktops, servers and supercomputers

- Reducing the code size is essential
  - chip area and cost
  - performance and power



IR Level vs. Binary Level

- Typical code size optimizations work on IR
  - similar code merging
  - dead-code elimination
  - outlining

- Limited on many scenarios
  - code size exceeding when binaries get deployed
  - no source code provided (assembly)
  - hard to be recompiled (legacy code)

IR Level vs. Binary Level

- Typical code size optimizations work on IR
  - similar code merging
  - dead-code elimination
  - outlining

- Limited on many scenarios
  - code size exceeding when binaries get deployed
  - no source code provided (assembly)
  - hard to be recompiled (legacy code)

How do we reduce code size at Binary Level?

Outline

- Introduction
- Background and Motivation
- RollBin Design
- Evaluation
- Summary

Loop Unrolling/Rerolling

- Loops are predominated in almost all programs
  - are optimization focus of the compiler

Loop Unrolling/Rerolling

- Loops are predominated in almost all programs
  - are optimization focus of the compiler
- Loop unrolling
  - replicates the loop body multiple times
  - enhances a program's execution
  - enables further optimizations
  - increases code size

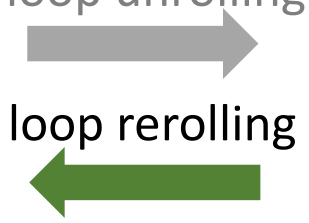
```
for(i=0; i<100; i++)
    x[i]=x[i]+s;
```


```
for(i=0; i<100; i+=4) {
    x[i]=x[i]+s;
    x[i+1]=x[i+1]+s;
    x[i+2]=x[i+2]+s;
    x[i+3]=x[i+3]+s;
}
```

Loop Unrolling/Rerolling

- Loops are predominated in almost all programs
  - are optimization focus of the compiler
- Loop unrolling
  - replicates the loop body multiple times
  - enhances a program's execution
  - enables further optimizations
  - increases code size
- Loop rerolling
  - transforms an unrolled loop into a rolled one

```
for(i=0; i<100; i++)
    x[i]=x[i]+s;
```

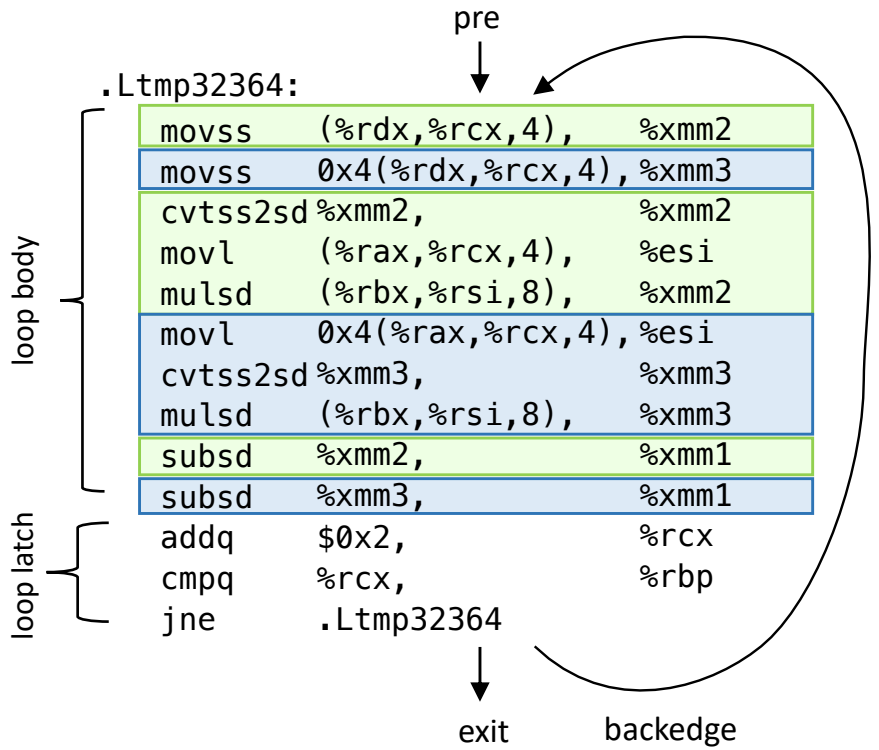



```
for(i=0; i<100; i+=4) {
    x[i]=x[i]+s;
    x[i+1]=x[i+1]+s;
    x[i+2]=x[i+2]+s;
    x[i+3]=x[i+3]+s;
}
```

An Example

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

loop unrolling



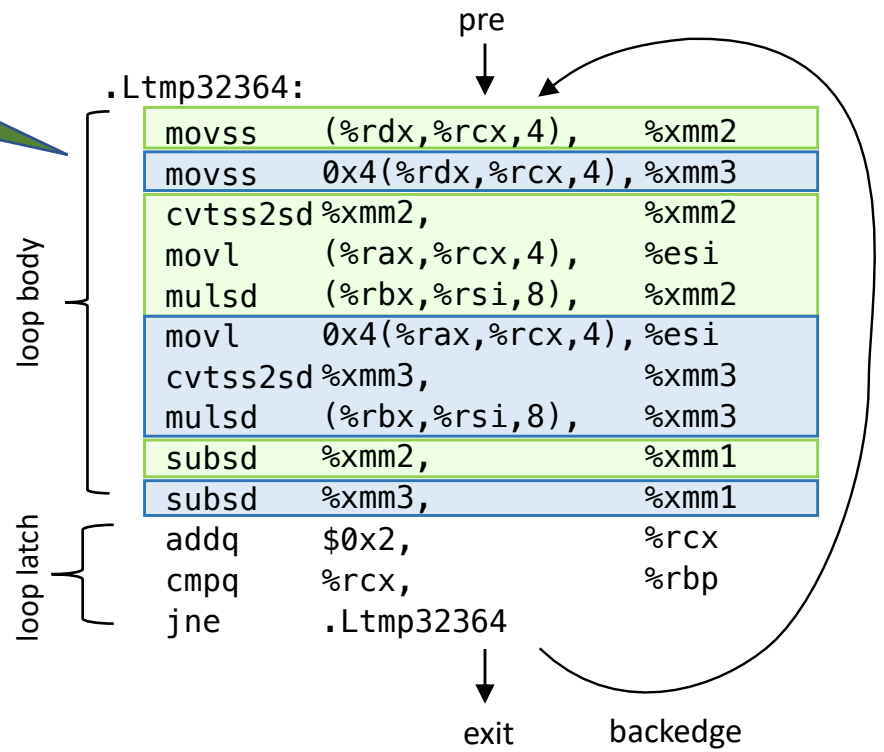
*from SPEC2006 482.sphinx3

An Example

a **62.5%** increase
(8 instructions to 13)

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

loop unrolling



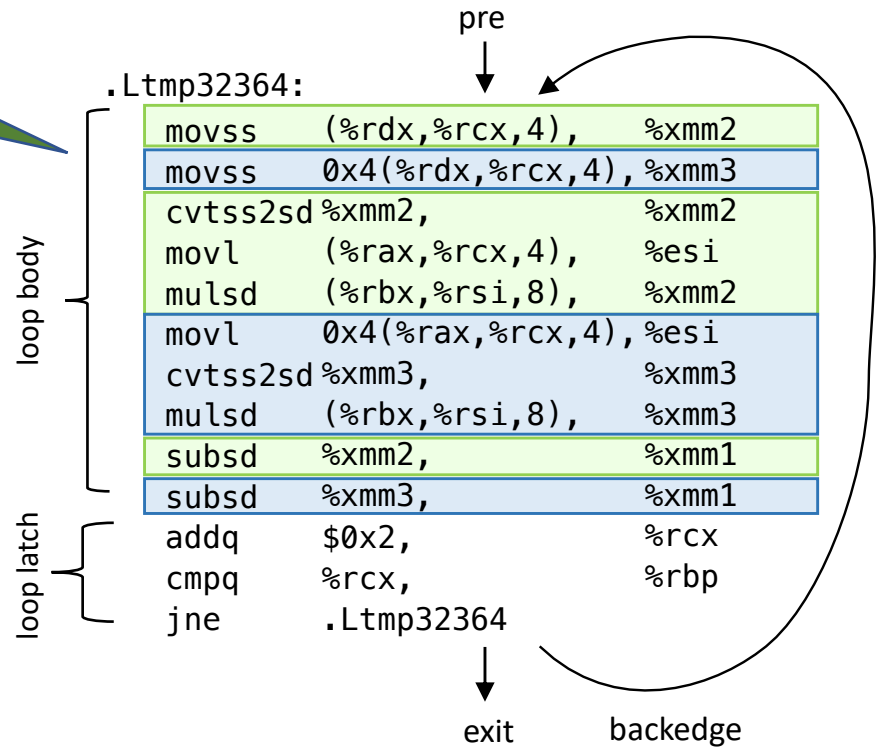
*from SPEC2006 482.sphinx3

An Example

a **62.5%** increase
(8 instructions to 13)

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

loop unrolling



*from SPEC2006 482.sphinx3

3.2% - 36.7% increase in SPEC and MiBench benchmark suites

Challenges

Challenges

- How to retrieve critical clues of loops from **raw** instructions?

raw instructions

```
.Ltmp32364:
    movss    (%rdx,%rcx,4),    %xmm2
    movss    0x4(%rdx,%rcx,4), %xmm3
    cvtss2sd %xmm2,            %xmm2
    movl     (%rax,%rcx,4),    %esi
    mulsd    (%rbx,%rsi,8),    %xmm2
    movl     0x4(%rax,%rcx,4), %esi
    cvtss2sd %xmm3,            %xmm3
    mulsd    (%rbx,%rsi,8),    %xmm3
    subsd    %xmm2,            %xmm1
    subsd    %xmm3,            %xmm1
    addq     $0x2,             %rcx
    cmpq     %rcx,             %rbp
    jne      .Ltmp32364
```

Challenges

- How to retrieve critical clues of loops from **raw** instructions?
- How to differentiate the **reordered** instructions from multiple iterations?

raw and **reordered** instructions

```
.Ltmp32364:
    movss    (%rdx,%rcx,4),    %xmm2
    movss    0x4(%rdx,%rcx,4), %xmm3
    cvtss2sd %xmm2,            %xmm2
    movl     (%rax,%rcx,4),    %esi
    mulsd    (%rbx,%rsi,8),    %xmm2
    movl     0x4(%rax,%rcx,4), %esi
    cvtss2sd %xmm3,            %xmm3
    mulsd    (%rbx,%rsi,8),    %xmm3
    subsd    %xmm2,            %xmm1
    subsd    %xmm3,            %xmm1
    addq     $0x2,             %rcx
    cmpq     %rcx,             %rbp
    jne      .Ltmp32364
```

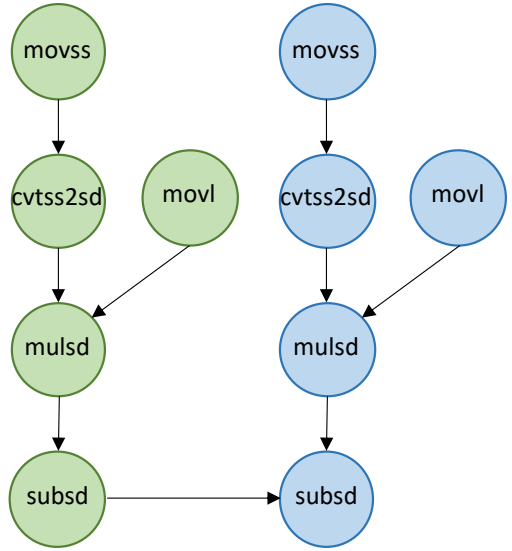
Challenges

- How to retrieve critical clues of loops from **raw** instructions?
- How to differentiate the **reordered** instructions from multiple iterations?
- How to handle the **across-iteration** data dependency?

raw and **reordered** instructions

```
.Ltmp32364:
    movss    (%rdx,%rcx,4),    %xmm2
    movss    0x4(%rdx,%rcx,4), %xmm3
    cvtss2sd %xmm2,            %xmm2
    movl     (%rax,%rcx,4),    %esi
    mulsd    (%rbx,%rsi,8),    %xmm2
    movl     0x4(%rax,%rcx,4), %esi
    cvtss2sd %xmm3,            %xmm3
    mulsd    (%rbx,%rsi,8),    %xmm3
    subsd    %xmm2,            %xmm1
    subsd    %xmm3,            %xmm1
    addq     $0x2,             %rcx
    cmpq     %rcx,             %rbp
    jne      .Ltmp32364
```

across-iteration dependency



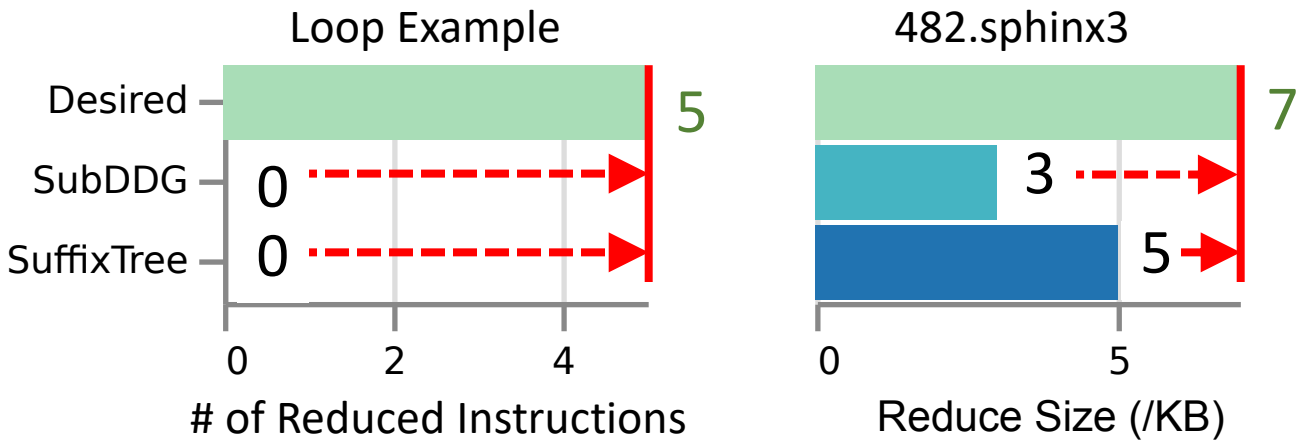
Previous Efforts

- SubDDG [ICIS'2015]
  - uses data dependency graph to partition iterations
  - operates on independent loops
- SuffixTree [ICCAD'2005]
  - uses suffix trees to identify repeated instructions
  - benefits loops with repetitive instruction structure

[1] Erh-Wen Hu, Bogong Su, and Jian Wang. 2016. Instruction Level Loop De-optimization. In Computer and Information Science 2015. 221–234
[2] Greg Stiff and Frank Vahid. 2005. New Decompilation Techniques for Binary-Level Co-Processor Generation. In Proceedings of the 2005 International Conference on Computer-Aided Design . 547–554.

Previous Efforts

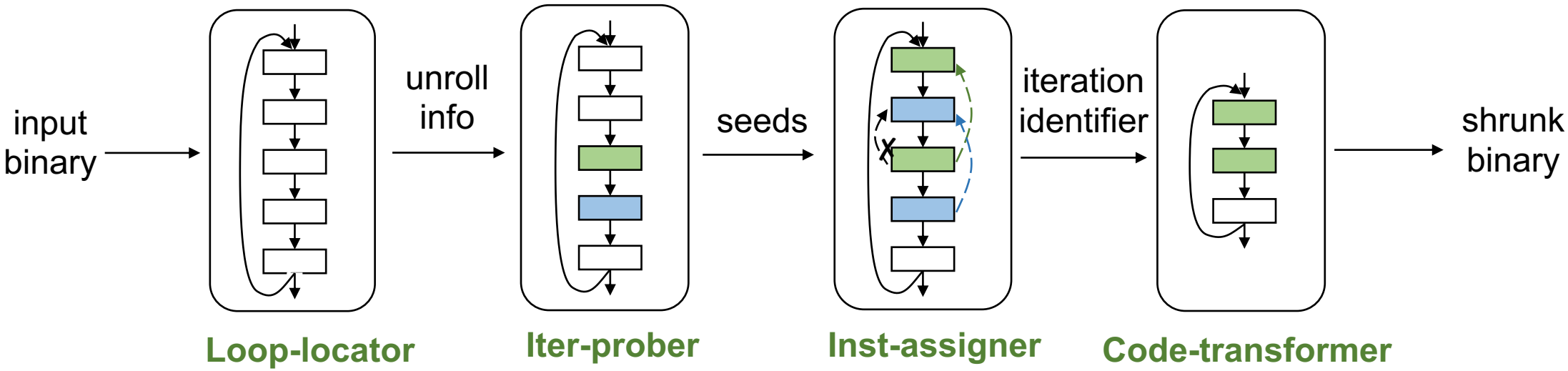
- SubDDG [ICIS'2015]
  - uses data dependency graph to partition iterations
  - operates on independent loops
- SuffixTree [ICCAD'2005]
  - uses suffix trees to identify repeated instructions
  - benefits loops with repetitive instruction structure



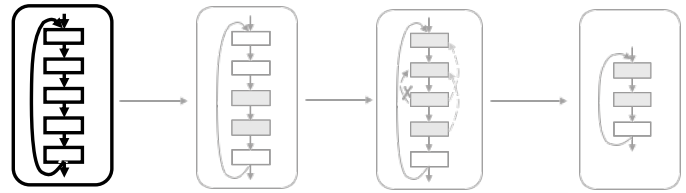
[1] Erh-Wen Hu, Bogong Su, and Jian Wang. 2016. Instruction Level Loop De-optimization. In Computer and Information Science 2015. 221–234
[2] Greg Stiff and Frank Vahid. 2005. New Decompilation Techniques for Binary-Level Co-Processor Generation. In Proceedings of the 2005 International Conference on Computer-Aided Design . 547–554.

RollBin: Loop Rerolling at Binary Level

- Four steps
  - **Loop-locator** to identify the loops
  - **Iter-prober** to anchor the iterations
  - **Inst-assigner** to assign each instruction to its iteration
  - **Code-transforme**r to revise the code



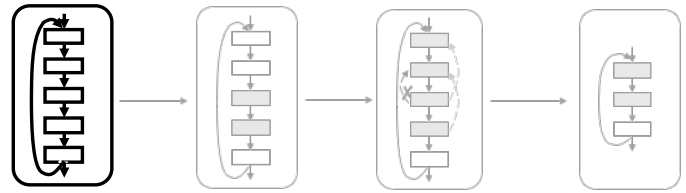
Identify Unrolled Loops



1. detect the loop
2. backtrack the associated memory addressing

```
.Ltmp32364:
    movss    (%rdx,%rcx,4),    %xmm2
    movss    0x4(%rdx,%rcx,4), %xmm3
    cvtss2sd %xmm2,            %xmm2
    movl     (%rax,%rcx,4),    %esi
    mulsd    (%rbx,%rsi,8),    %xmm2
    movl     0x4(%rax,%rcx,4), %esi
    cvtss2sd %xmm3,            %xmm3
    mulsd    (%rbx,%rsi,8),    %xmm3
    subsd    %xmm2,            %xmm1
    subsd    %xmm3,            %xmm1
    addq     $0x2,             %rcx
    cmpq     %rcx,             %rbp
    jne      .Ltmp32364
```

Identify Unrolled Loops

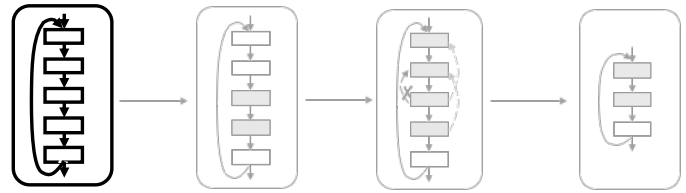


1. detect the loop
2. backtrack the associated memory addressing

```
.Ltmp32364:
    movss    (%rdx,%rcx,4),    %xmm2
    movss    0x4(%rdx,%rcx,4), %xmm3
    cvtss2sd %xmm2,            %xmm2
    movl     (%rax,%rcx,4),    %esi
    mulsd    (%rbx,%rsi,8),    %xmm2
    movl     0x4(%rax,%rcx,4), %esi
    cvtss2sd %xmm3,            %xmm3
    mulsd    (%rbx,%rsi,8),    %xmm3
    subsd    %xmm2,            %xmm1
    subsd    %xmm3,            %xmm1
    addq     $0x2,             %
    cmpq     %rcx,             %rbp
    jne      .Ltmp32364
```

induction register

Identify Unrolled Loops



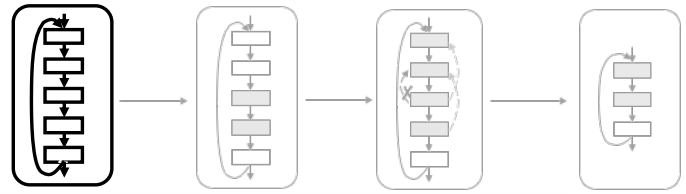
1. detect the loop
2. backtrack the associated memory addressing

```
.Ltmp32364:
    movss     (%rdx,%rcx,4),     %xmm2
    movss     0x4(%rdx,%rcx,4),  %xmm3
    cvtss2sd  %xmm2,             %xmm2
    movl      (%rax,%rcx,4),     %esi
    mulsd     (%rbx,%rsi,8),     %xmm2
    movl      0x4(%rax,%rcx,4),  %esi
    cvtss2sd  %xmm3,             %xmm3
    mulsd     (%rbx,%rsi,8),     %xmm3
    subsd     %xmm2,             %xmm1
    subsd     %xmm3,             %xmm1
    addq      $0x2,              %rcx
    cmpq      %rcx,              %rbp
    jne       .Ltmp32364
```

memory addresses related to the induction register

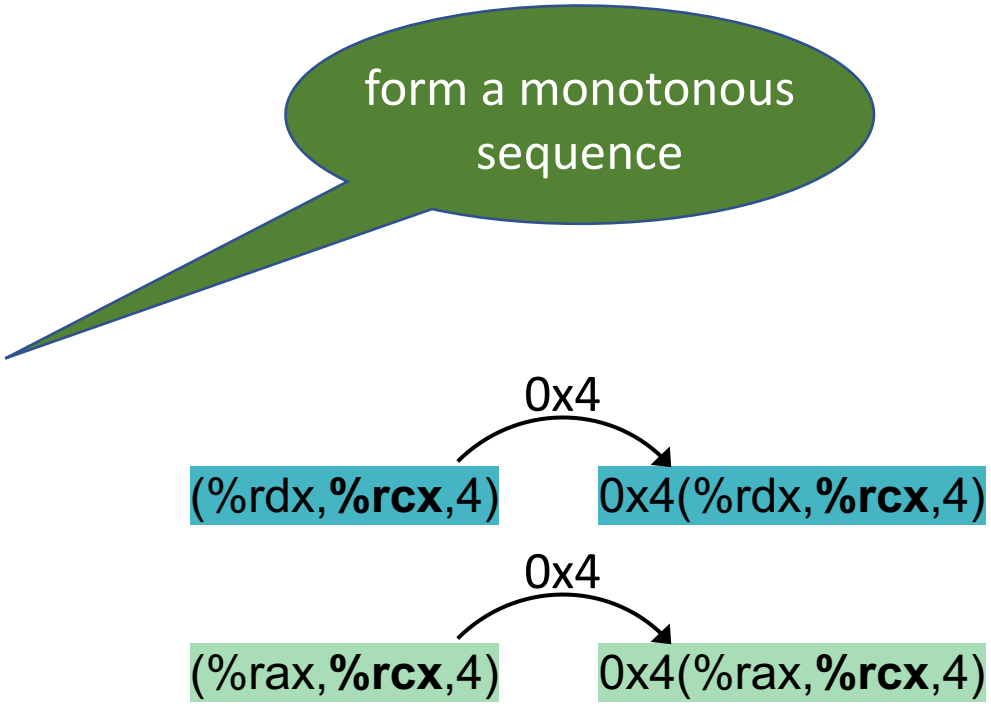
induction register

Identify Unrolled Loops

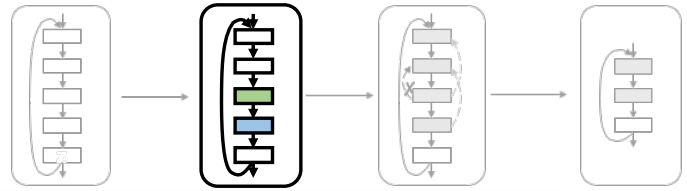


1. detect the loop
2. backtrack the associated memory addressing

```
.Ltmp32364:
    movss     (%rdx,%rcx,4),     %xmm2
    movss     0x4(%rdx,%rcx,4),  %xmm3
    cvtss2sd  %xmm2,             %xmm2
    movl      (%rax,%rcx,4),     %esi
    mulsd     (%rbx,%rsi,8),     %xmm2
    movl      0x4(%rax,%rcx,4),  %esi
    cvtss2sd  %xmm3,             %xmm3
    mulsd     (%rbx,%rsi,8),     %xmm3
    subsd     %xmm2,             %xmm1
    subsd     %xmm3,             %xmm1
    addq      $0x2,              %rcx
    cmpq      %rcx,              %rbp
    jne       .Ltmp32364
```



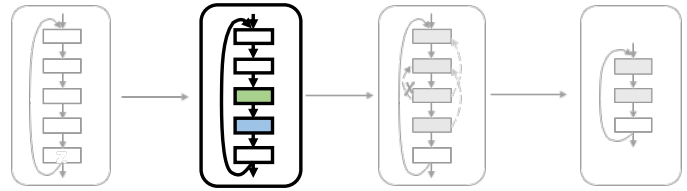
Anchor the Iterations



1. assign the iteration number to induction-related instructions
2. mark them as **seed** instructions

```
.Ltmp32364:
  movss    (%rdx,%rcx,4),    %xmm2     #(0-0)/4=0 → iter 0
  movss    0x4(%rdx,%rcx,4), %xmm3     #(4-0)/4=1 → iter 1
  cvtss2sd %xmm2,            %xmm2
  movl     (%rax,%rcx,4),    %esi      #(0-0)/4=0 → iter 0
  mulsd    (%rbx,%rsi,8),    %xmm2
  movl     0x4(%rax,%rcx,4), %esi      #(4-0)/4=1 → iter 1
  cvtss2sd %xmm3,            %xmm3
  mulsd    (%rbx,%rsi,8),    %xmm3
  subsd    %xmm2,            %xmm1
  subsd    %xmm3,            %xmm1
  addq     $0x2,             %rcx
  cmpq     %rcx,             %rbp
  jne      .Ltmp32364
```

Anchor the Iterations



1. assign the iteration number to induction-related instructions
2. mark them as **seed** instructions

```
.Ltmp32364:
  movss    (%rdx,%rcx,4),    %xmm2
  movss    0x4(%rdx,%rcx,4), %xmm3
  cvtss2sd %xmm2,            %xmm2
  movl     (%rax,%rcx,4),    %esi
  mulsd    (%rbx,%rsi,8),    %xmm2
  movl     0x4(%rax,%rcx,4), %esi
  cvtss2sd %xmm3,            %xmm3
  mulsd    (%rbx,%rsi,8),    %xmm3
  subsd    %xmm2,            %xmm1
  subsd    %xmm3,            %xmm1
  addq     $0x2,             %rcx
  cmpq     %rcx,             %rbp
  jne      .Ltmp32364
```

#(0-0)/4=0 → *iter 0*

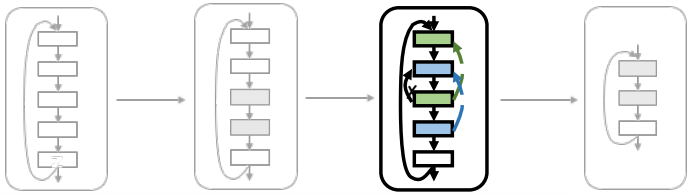
#(4-0)/4=1 → *iter 1*

#(0-0)/4=0 → *iter 0*

#(4-0)/4=1 → *iter 1*

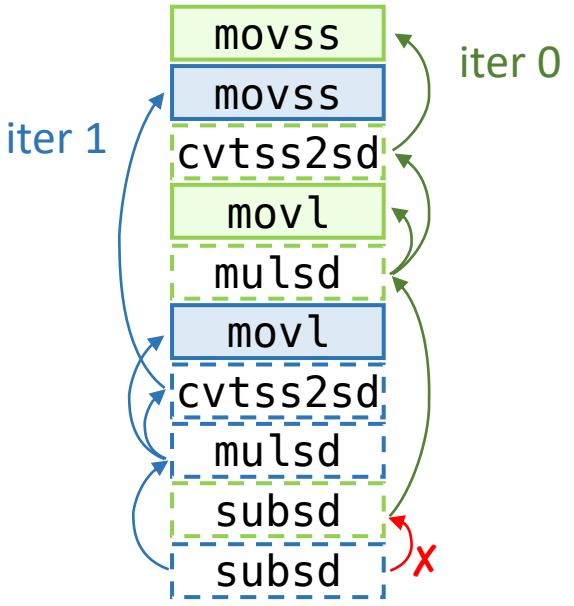
Seed instructions: communicate the original information of iterations

Cluster the Instructions

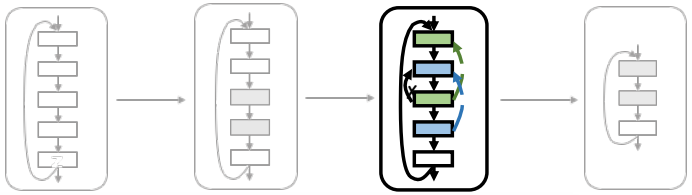


1. propagate the iteration number along the data dependency
2. intercept the undesired number propagation by **wall** instructions

```
.Ltmp32364:
  movss    (%rdx,%rcx,4),    %xmm2
  movss    0x4(%rdx,%rcx,4), %xmm3
  cvtss2sd %xmm2,            %xmm2
  movl     (%rax,%rcx,4),    %esi
  mulsd    (%rbx,%rsi,8),    %xmm2
  movl     0x4(%rax,%rcx,4), %esi
  cvtss2sd %xmm3,            %xmm3
  mulsd    (%rbx,%rsi,8),    %xmm3
  subsd    %xmm2,            %xmm1
  subsd    %xmm3,            %xmm1
  addq     $0x2,             %rcx
  cmpq     %rcx,             %rbp
  jne      .Ltmp32364
```

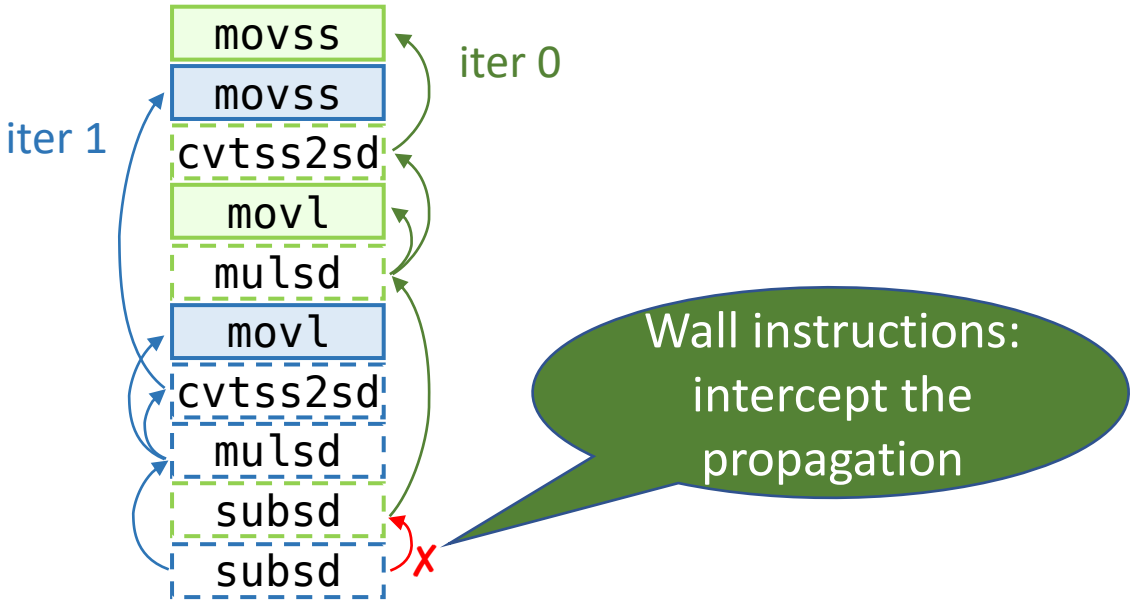


Cluster the Instructions

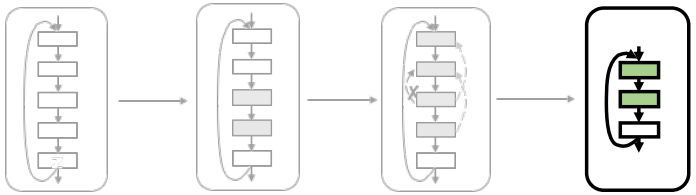


1. propagate the iteration number along the data dependency
2. intercept the undesired number propagation by **wall** instructions

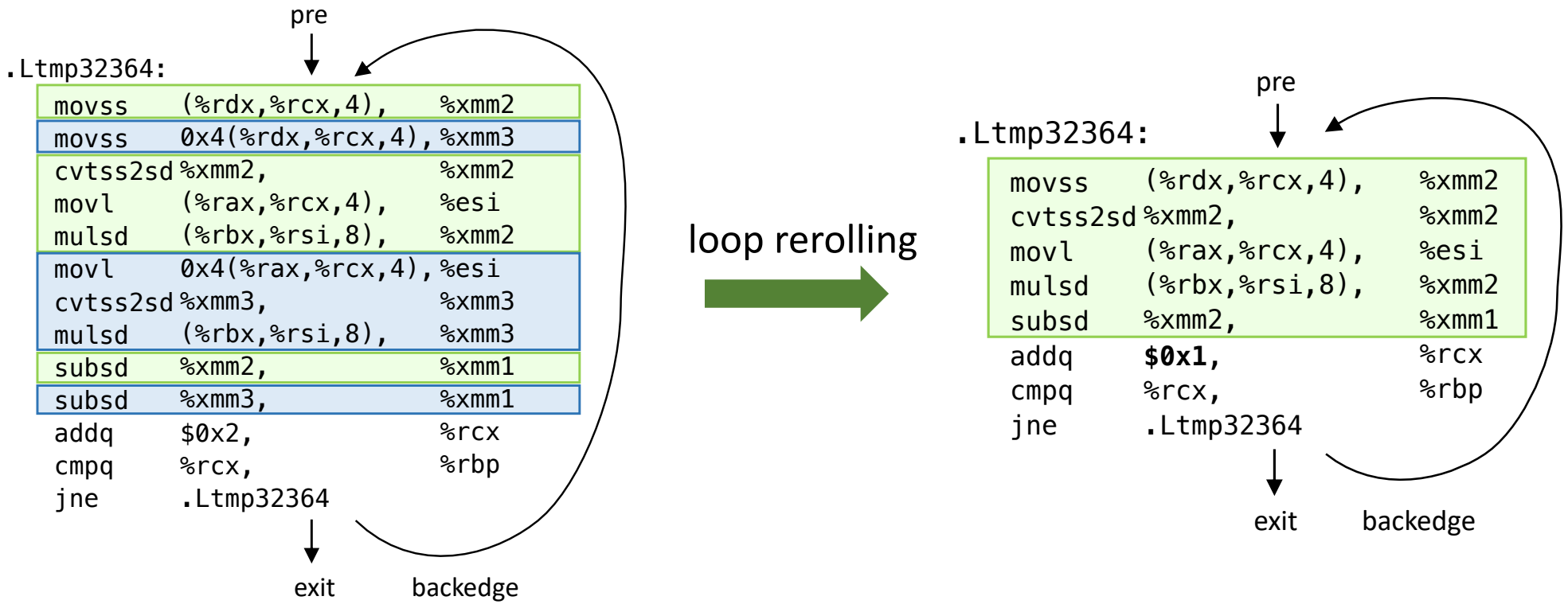
```
.Ltmp32364:
  movss    (%rdx,%rcx,4),    %xmm2
  movss    0x4(%rdx,%rcx,4), %xmm3
  cvtss2sd %xmm2,            %xmm2
  movl     (%rax,%rcx,4),    %esi
  mulsd    (%rbx,%rsi,8),    %xmm2
  movl     0x4(%rax,%rcx,4), %esi
  cvtss2sd %xmm3,            %xmm3
  mulsd    (%rbx,%rsi,8),    %xmm3
  subsd    %xmm2,            %xmm1
  subsd    %xmm3,            %xmm1
  addq     $0x2,             %rcx
  cmpq     %rcx,             %rbp
  jne      .Ltmp32364
```



Transform the Code

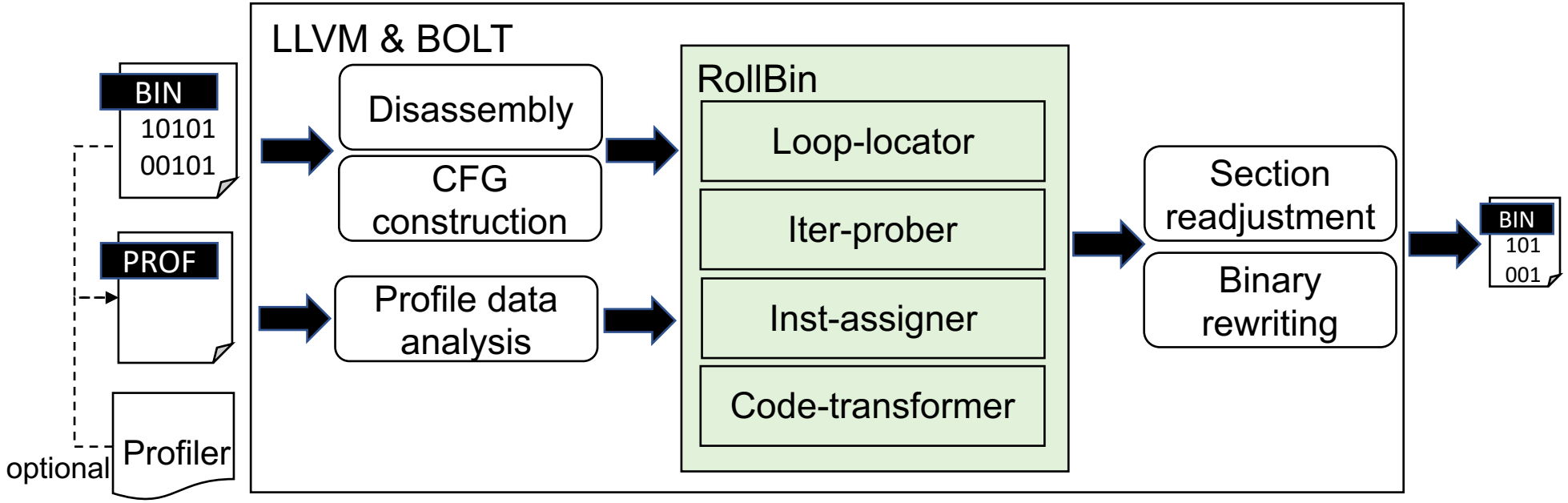


1. remove iteration and update latch
2. alleviate performance degradation with profiling data



Implementation

- Base on LLVM and BOLT[CGO'2019]
  - add "BinaryLoopRerolling" pass
  - reimplement the binary rewriting module
    - adjust section offsets for the final executable size



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

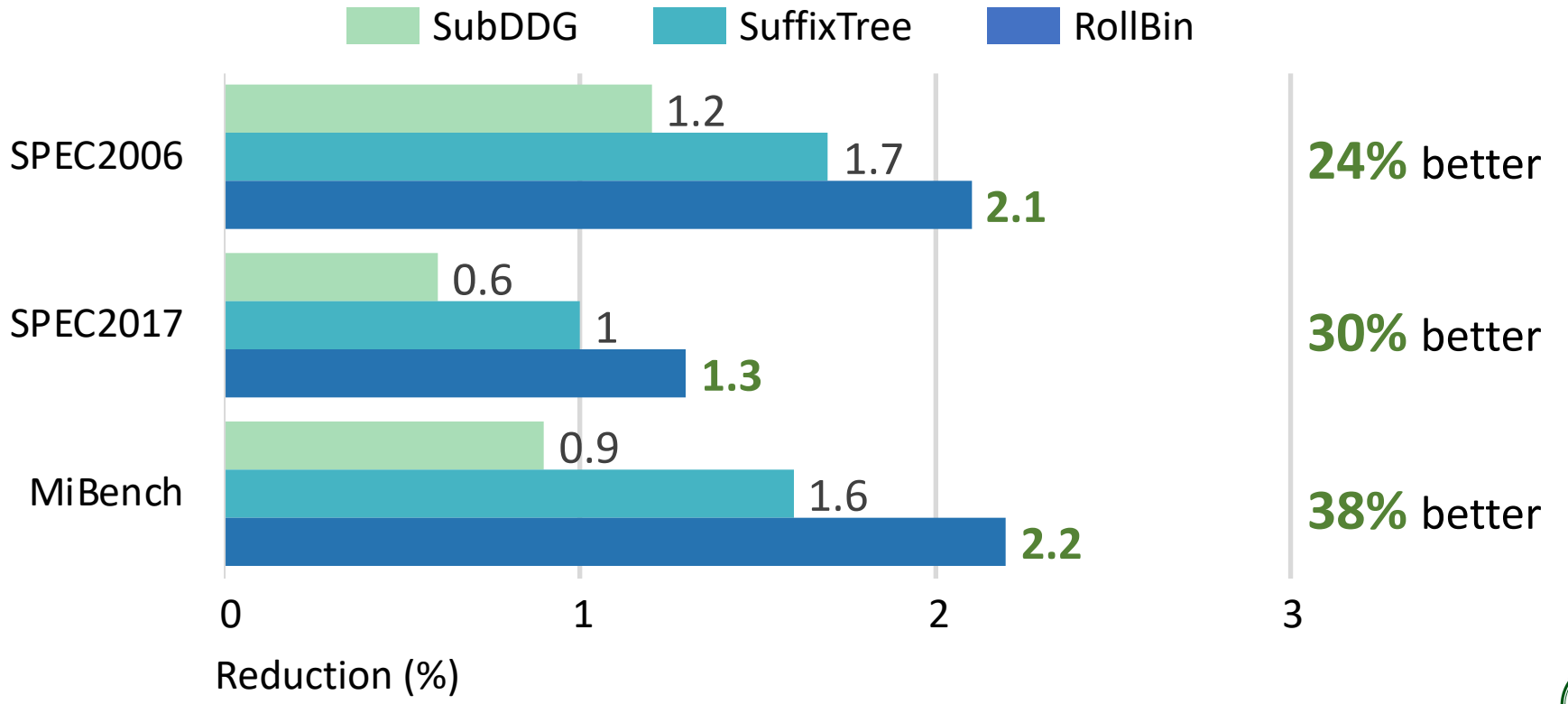
Evaluation Setup

- Hardware & Software
  - AMD EPYC 7742 CPU
  - LLVM 13.0.0
- Build flag
  - O2, O3, and Os with loop unrolling being enabled
- Contending designs
  - RollBin, SubDDG, SuffixTree, RoLAG[CGO'2022]
- Metrics
  - code size reduction: the size of .text segment
  - performance: execution time

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

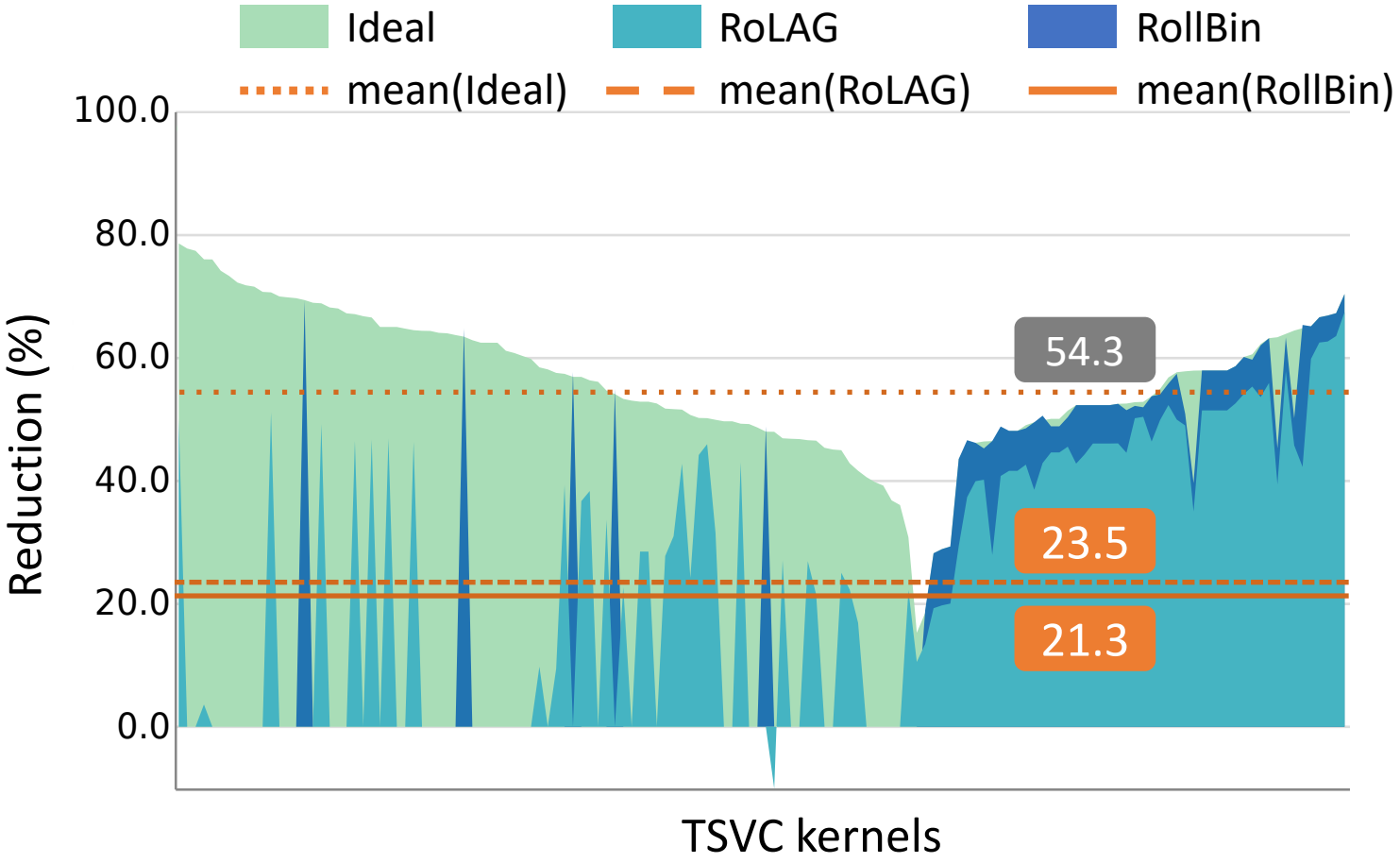
Code-size

- RollBin beats SubDDG and SuffixTree
  - effectively shrinks code size by **2.1%**, **1.7%** and **2.2%** on average
  - reduces total code size by **173** KB, **428** KB and **65** KB



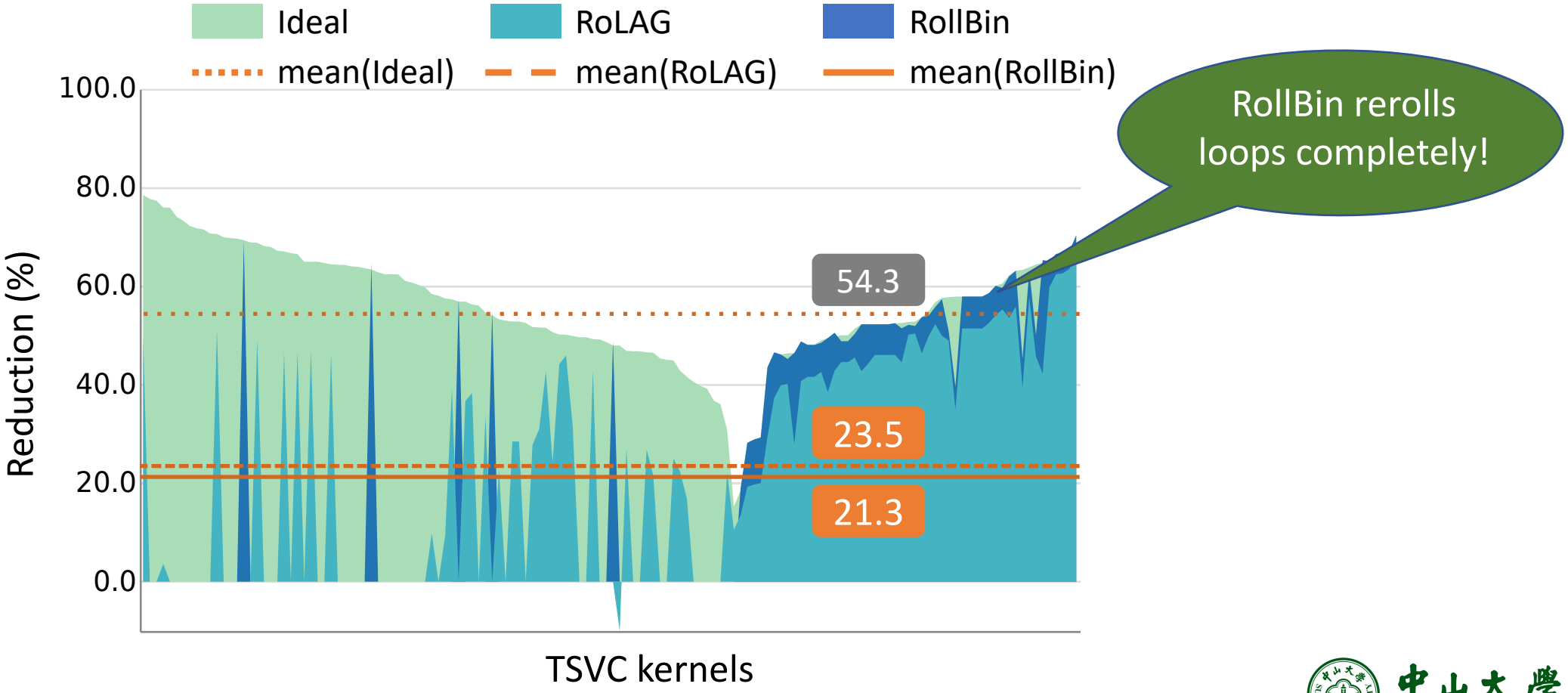
Compare to IR-level Rerolling

- RollBin achieves similar reductions with RoLAG
  - **without** source code level information



Compare to IR-level Rerolling

- RollBin achieves similar reductions with RoLAG
  - **without** source code level information



Real Apps

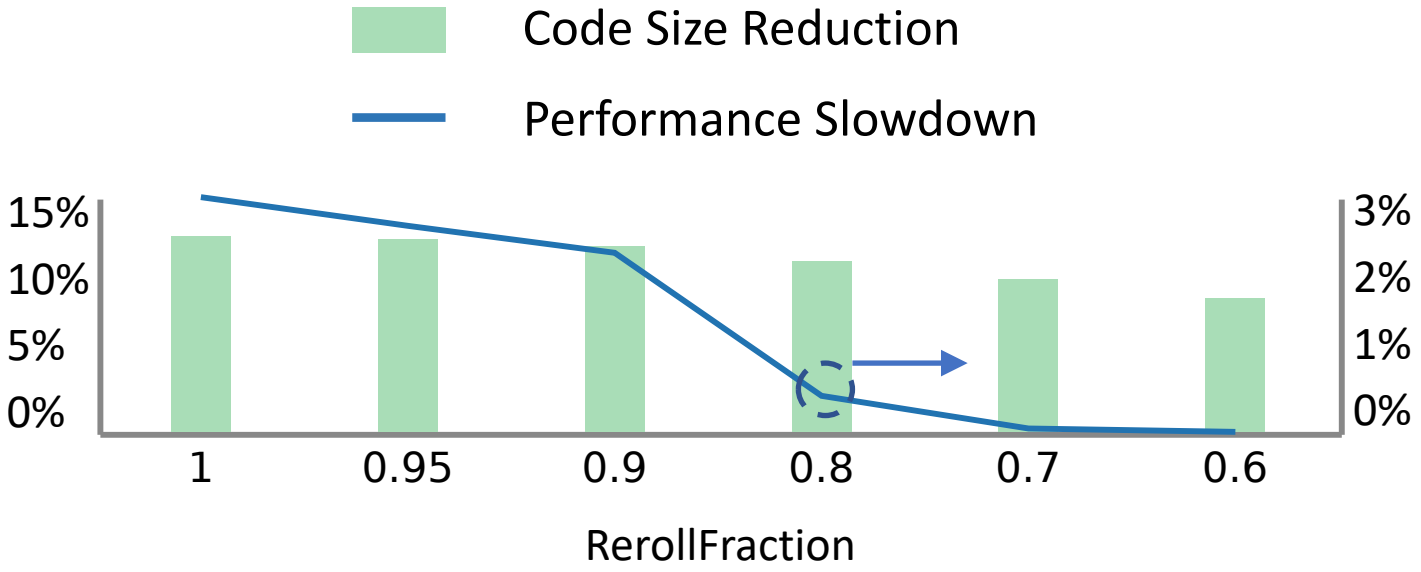
- TensorFLow Lite[1]
  - a lightweight deep learning framework for mobile devices
  - contains rich **machine learning layers** like convolution and pooling
  - code size reduced by **81 KB** or by **1.9%** over **2024** unrolled loops

- BLASFEO[2]
  - a library of BLAS and LAPACK-like routines optimized for embedded infrastructures
  - employs plenty of **handcrafted assembly-coded** dense linear algebra kernels
  - code size reduced by **24 KB** or by **1.6%** over **669** unrolled loops

[1] TensorFlow. 2022. ML for Mobile and Edge Devices - TensorFlow Lite. https://www.tensorflow.org/lite
[2] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2018. BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization. ACM Trans. Math. Softw. 44, 4 (2018), 42:1–42:30

Performance Tradeoff

- RollBin allows to balance code size and performance though RerollFraction knob
  - lowers the performance slowdown from **3%** to **1%** on TSVC
  - maintains comparable size reduction of **12%**



Summary

- We highlight the critical need of performing **binary level optimizations** to reduce code size

- A novel design **RollBin** to reroll loops at binary level
  - recognizes iterations by identifying regular memory address patterns
  - reconstructs the iterations using a customized data dependency analysis

- RollBin reduces code size effectively
  - outperforms the SOTAs on benchmark suites and real applications
  - uses profiling data to configure the trade-offs between code size and performance

Thanks!

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

Tianao Ge, Zewei Mo, Kan Wu, Xianwei Zhang, Yutong Lu

Sun Yat-sen University

Email: getianao@gmail.com

Backup

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

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