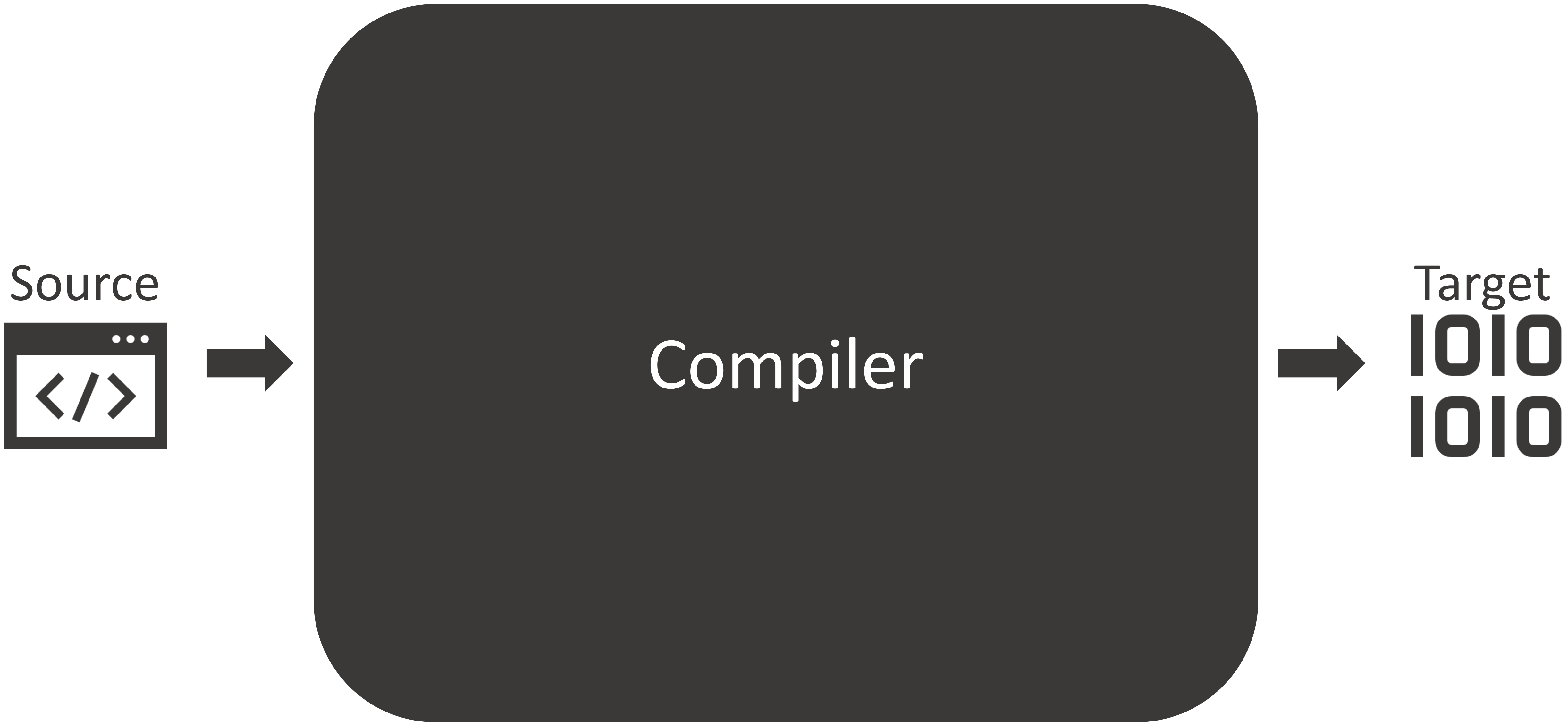
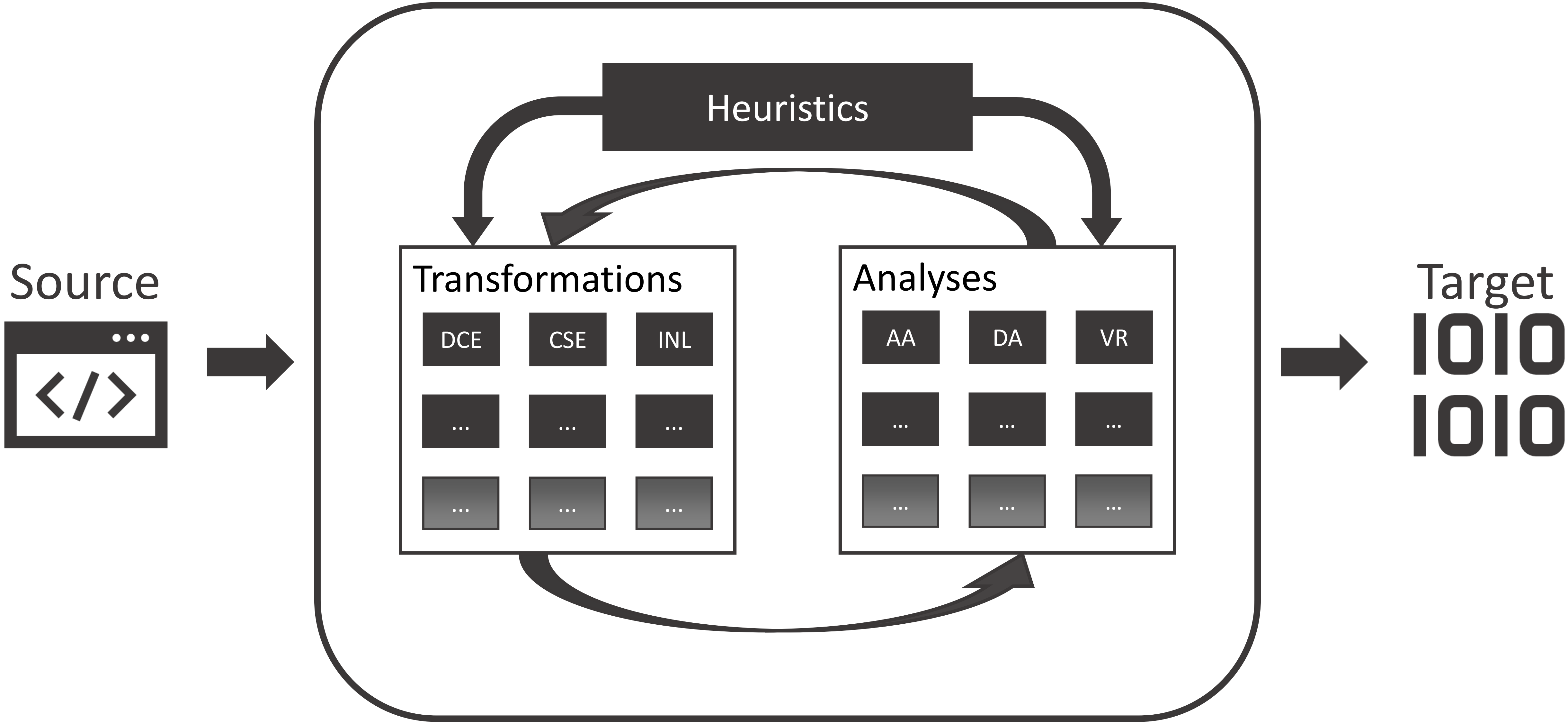
A Quest Toward the Perfect Optimizing Compiler

Theodoros Theodoridis

AST.ethz.ch
ADVANCED SOFTWARE TECHNOLOGIES

ETH zürich


Source
Compiler
Target
1010
1010


Source
Heuristics
Transformations
DCE
CSE
INL
...
Analyses
AA
DA
VR
...
Target
1010
1010

How far are we from the optimum? Pretty far…

BOLT: A Practical Binary Optimizer for Data Centers and Beyond

Maksim Panchenko, Rafael Auler, Bill Nell, Guilherme Ottoni
Facebook, Inc.
Menlo Park, CA, USA
{maks,rafaelauler,bnell,ottoni}@fb.com

Amir
@disruptnhandlr · **Follow**

I've been working on CMake magic to fully automatically apply BOLT to Clang, similar to how PGO two-stage build is automated with CMake cache file. It appears that fully automatically BOLT-optimized Clang is ~30% faster than the Release (-O3) Clang: **reviews.llvm.org/D132975**

3:08 AM · Sep 2, 2022

Read the full conversation on Twitter

166 Reply Share

Read 7 re

1.3x Speedup

timizations (FDO), also called *profile-guided optimizations* (PGO), particularly code layout. At the same time, due to their large sizes, applying FDO to these applications poses scalability challenges. Instrumentation-based profilers incur significant memory and computational performance costs, often making it impractical to gather accurate profiles from a production system. To simplify deployment and increase adoption, it is desirable to have a system that can obtain profile data for FDO from unmodified binaries running in their normal production environments. This is possible through the use of *sample-based profiling*, which enables high-quality profiles to be gathered with minimal operational complexity. This is the approach taken by tools such as Ispike [1], AutoFDO [2], and

resu… is particu… code layout.

We demonstrate the finding above … context of a static binary optimizer we built, called BOLT. BOLT is a modern, retargetable binary optimizer built on top of the LLVM compiler infrastructure [8]. Our experimental evaluation on large real-world applications shows that BOLT can improve performance by up to 20.4% on top of FDO and LTO. Furthermore, our analysis demonstrates that this improvement is mostly due to the improved code layout that is enabled by the more accurate usage of sample-based profile data at the binary level.

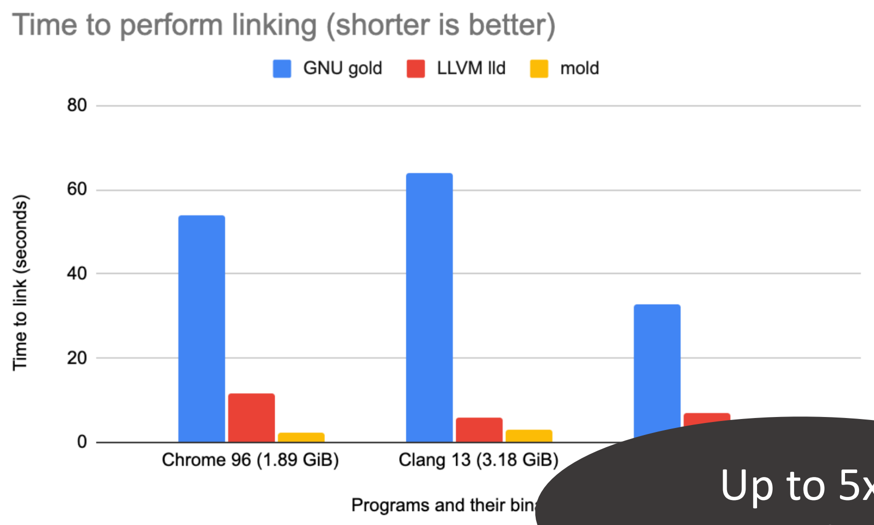
Overall, this paper makes the following contributions:

README.md

mold: A Modern Linker

mold is a faster drop-in replacement for existing Unix linkers. It is several times faster than the LLVM lld linker, the second-fastest open-source linker which I originally created a few years ago. mold is designed to increase developer productivity by reducing build time, especially in rapid debug-edit-rebuild cycles.

Here is a performance comparison of GNU gold, LLVM lld, and mold for linking final debuginfo-enabled executables of major large programs on a simulated 8-core 16-threads machine.

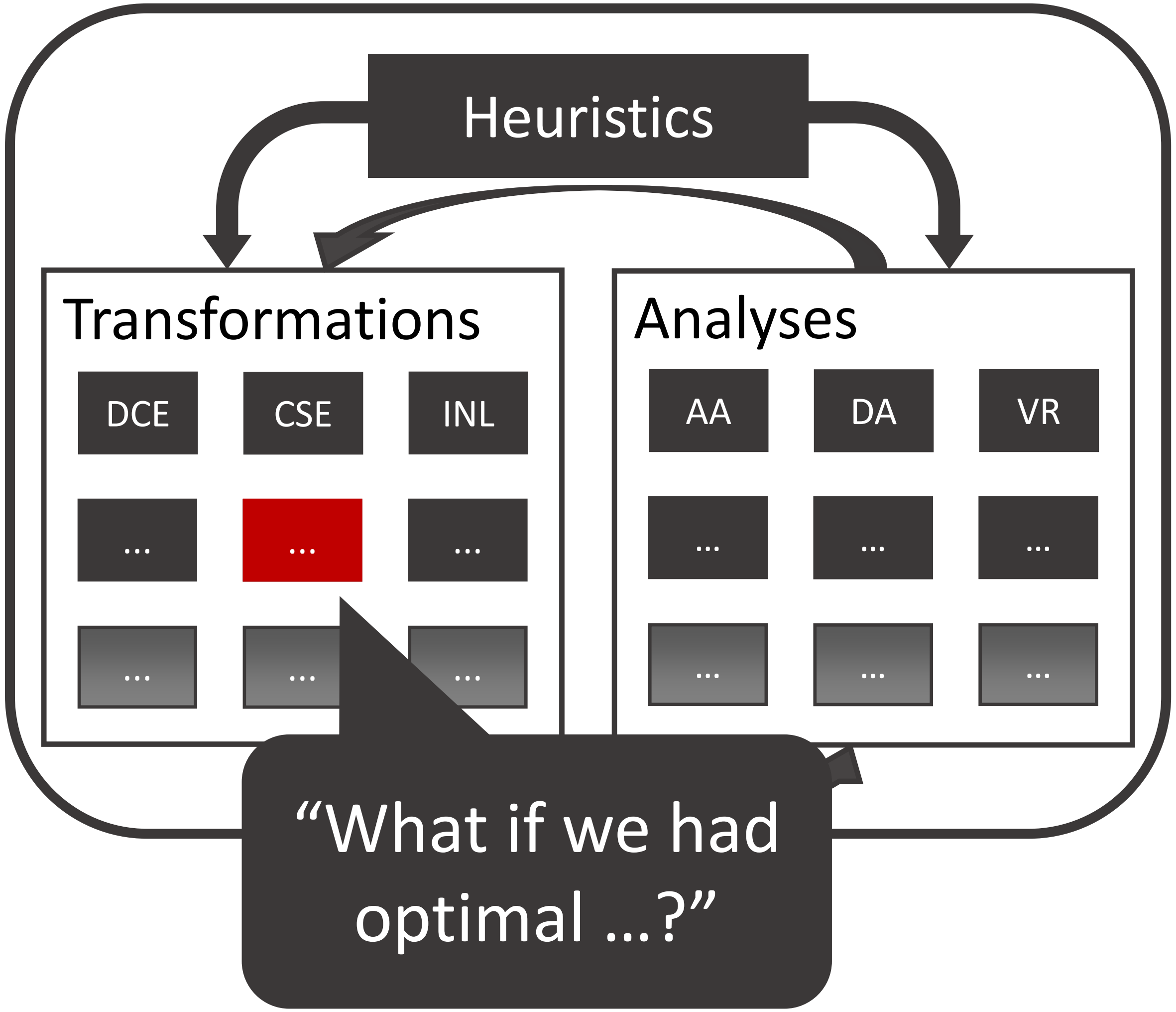


Up to 5x Speedup

Program (linker output size)	GNU gold	LLVM lld	mold
Chrome 96 (1.89 GiB)	53.86s	11.74s	2.21s
Clang 13 (3.18 GiB)	64.12s	5.82s	2.90s
Firefox 89 libxul (1.64 GiB)	32.95s	6.80s	1.42s

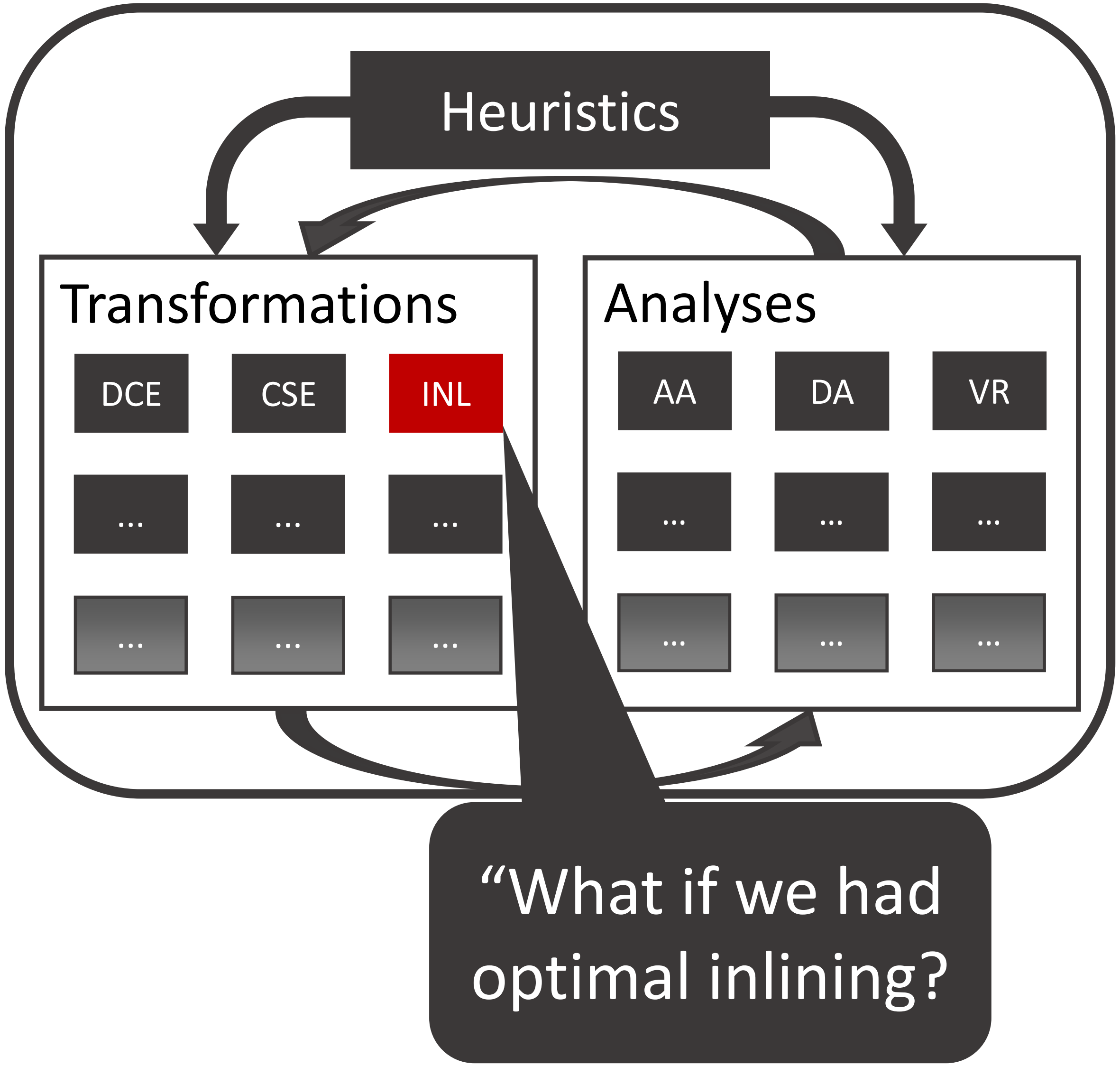
mold is so fast that it is only 2x *slower* than `cp` on the same machine. Feel free to file a bug if you find mold is not faster than other linkers.

Our Approach



1. We obtain the optimum.

2. We compare with the compiler and find the gap.



Understanding and Exploiting Optimal Function Inlining

Theodoros Theodoridis
theodoros.theodoridis@inf.ethz.ch
ETH Zurich
Switzerland

Tobias Grosser
tobias.grosser@ed.ac.uk
University of Edinburgh
United Kingdom

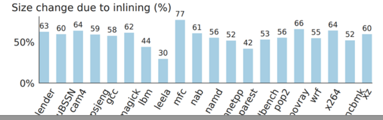
Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT

Inlining is a core transformation in optimizing compilers. It replaces a function call (call site) with the body of the called function (callee). It helps reduce function call overhead and binary size, and more importantly, enables other optimizations. The problem of inlining has been extensively studied, but it is far from being solved; predicting which inlining decisions are beneficial is nontrivial due to interactions with the rest of the compiler pipeline. Previous work has mainly focused on designing heuristics for better inlining decisions and has not investigated *optimal inlining*, *i.e.*, exhaustively finding the optimal inlining decisions. Optimal inlining is necessary for identifying and exploiting missed opportunities and evaluating the state of the art. This paper fills this gap through an extensive empirical analysis of optimal inlining using the SPEC2017 benchmark suite. Our novel formulation drastically reduces the inlining search space size (from 2^{349} down to 2^{25}) and allows us to exhaustively

1 INTRODUCTION

Function inlining (aka *inlining expansion*) is one of the fundamental compiler transformations. Not only does it eliminate function call overhead and potentially shrinks binary size, but it also expands the scope of intra-procedural analyses and optimizations. All of these are enabled by replacing function calls with the callees' bodies. The resulting optimization scope expansion makes inlining a critical transformation. Figure 1 illustrates the importance of inlining.



Finding Missed Optimizations through the Lens of Dead Code Elimination

Theodoros Theodoridis
theodoros.theodoridis@inf.ethz.ch
ETH Zurich
Switzerland

Manuel Rigger
manuel.rigger@inf.ethz.ch
ETH Zurich
Switzerland

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT

Compilers are foundational software development tools and incorporate increasingly sophisticated optimizations. Due to their complexity, it is difficult to systematically identify opportunities for improving them. Indeed, the automatic discovery of missed optimizations has been an important and significant challenge. The few existing approaches either cannot accurately pinpoint missed optimizations or target only specific analyses. This paper tackles this challenge by introducing a novel, effective approach that — in a simple and general manner — automatically identifies a wide range of missed optimizations. Our core insight is to leverage dead code elimination (DCE) to both analyze how well compilers optimize code and identify missed optimizations: (1) insert “optimization markers” in the basic blocks of a given program, (2) compute the

ACM Reference Format:
Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507764

1 INTRODUCTION

Both industry and academia have invested decades of effort to enhance compiler optimizations to improve the performance of computer programs [2, 3, 10, 16]. Despite these efforts, optimizing compilers are plagued by performance bugs, also known as missed optimization opportunities [24]. We define a missed optimization opportunity loosely as a case where a compiler produces

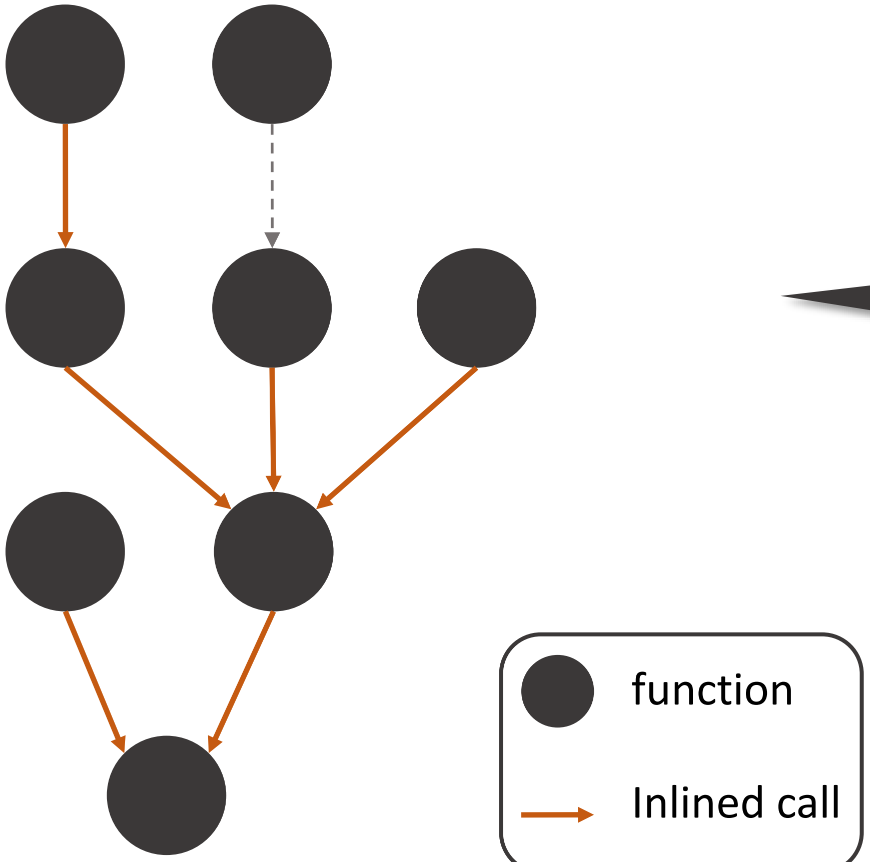
The Benefits of Inlining

```c
int bar(int a, int b) {
  if ((a * b) % 2)
    return a + b;
  else
    return a - b;
}


int foo(int x) {
  return bar(x,2) + 2;
}
```

after inlining →

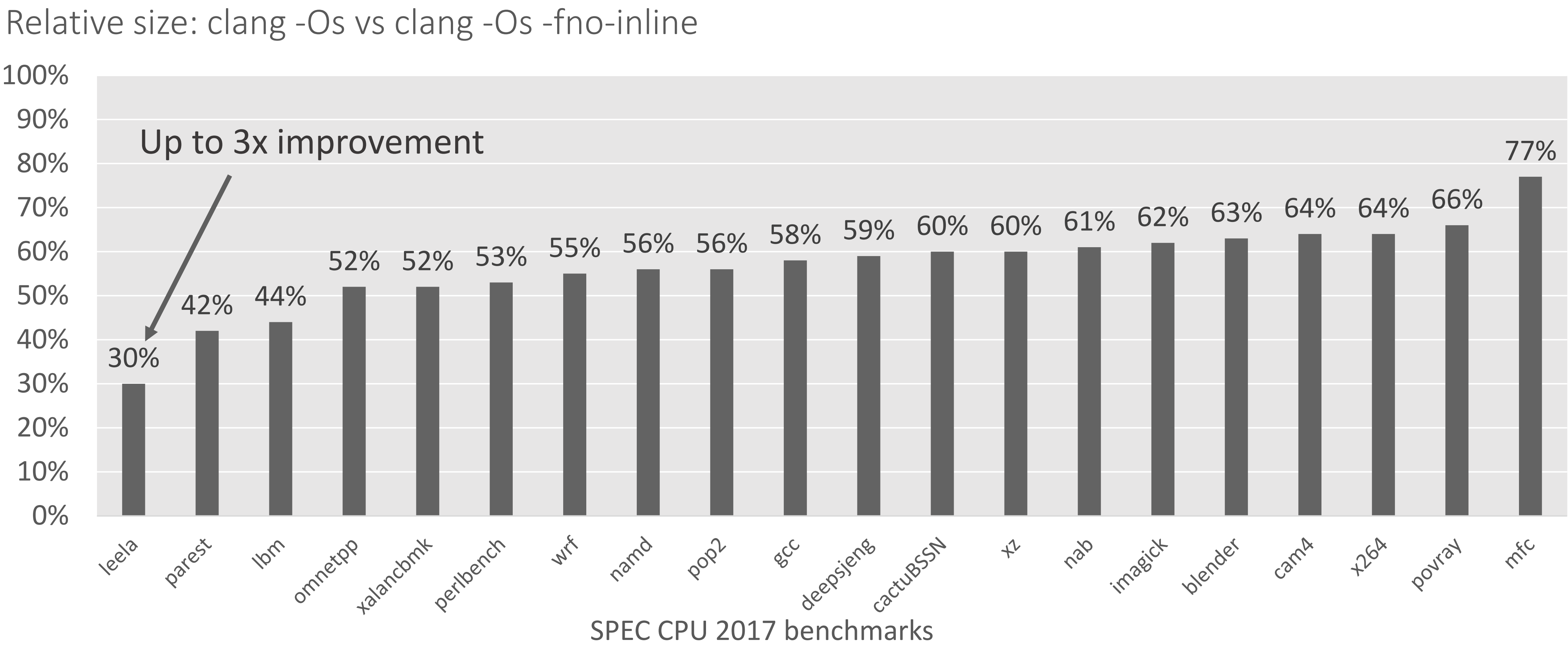
```c
int foo(int x) {
  return x;
}
```

Too much Inlining is Bad



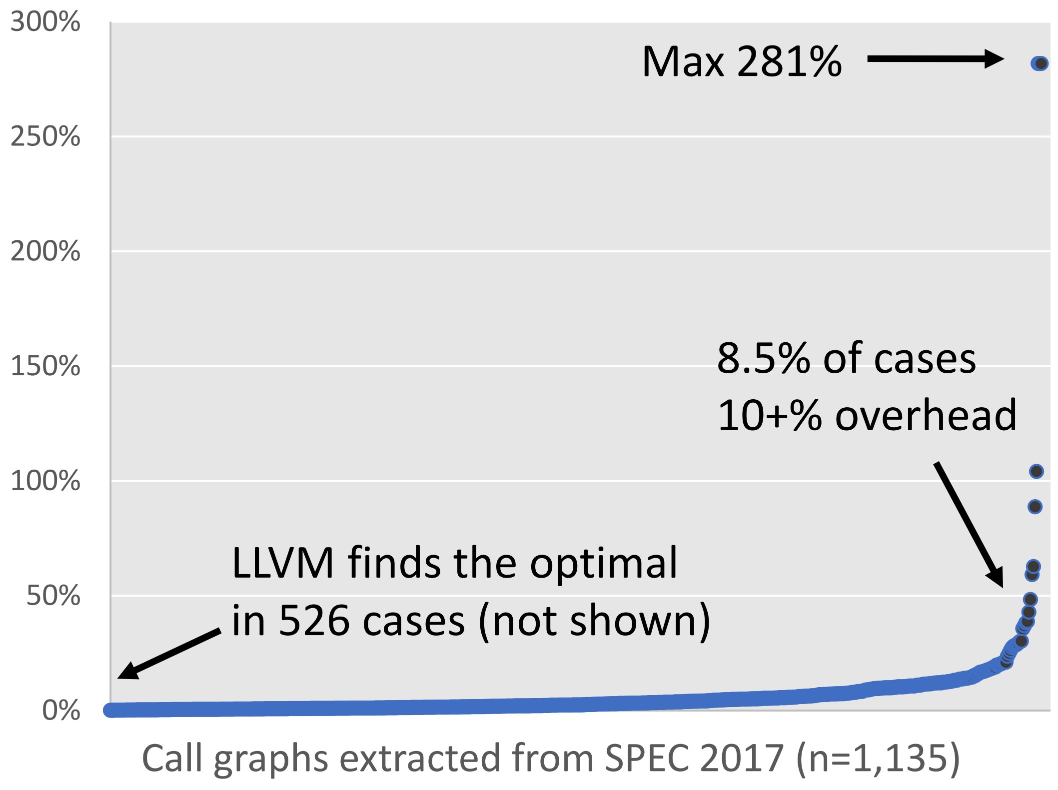
Aggressive Inlining:
69% binary size increase

Proper Inlining Reduces Program Size

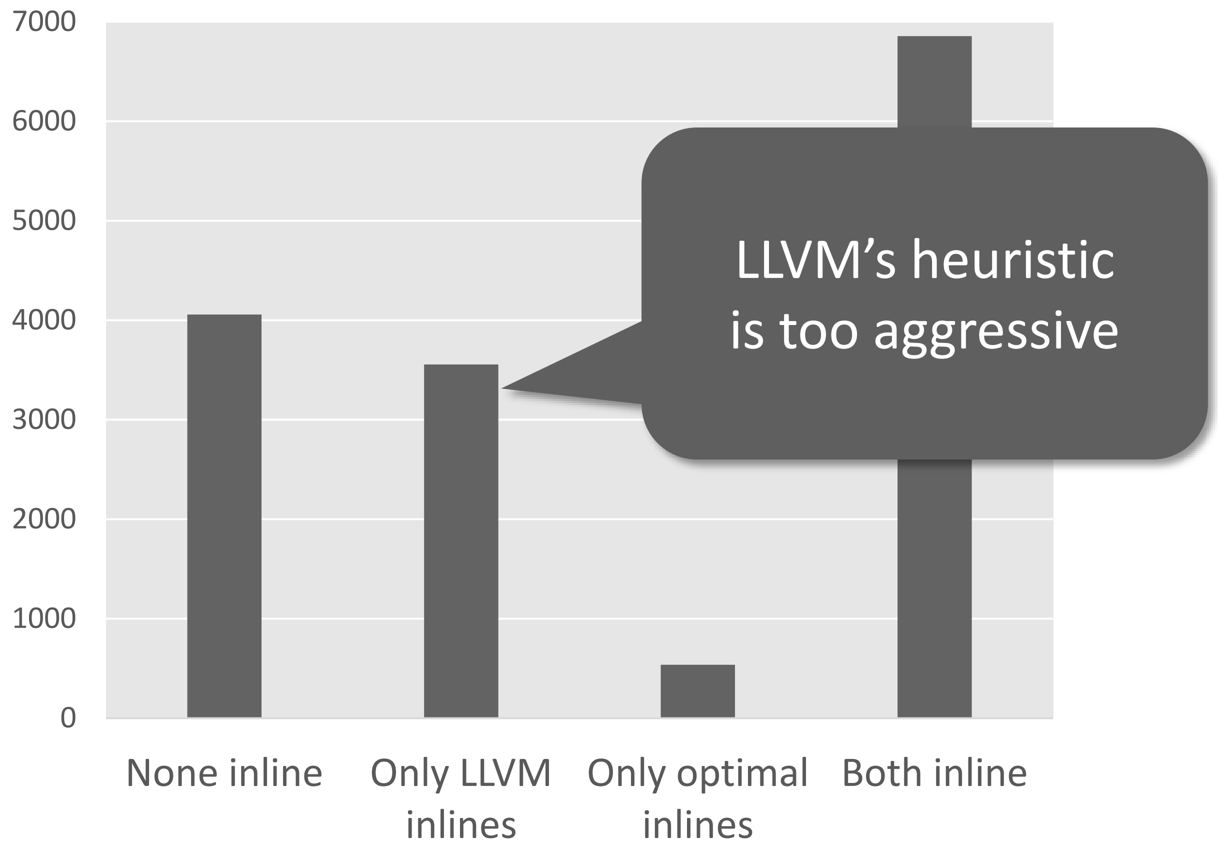


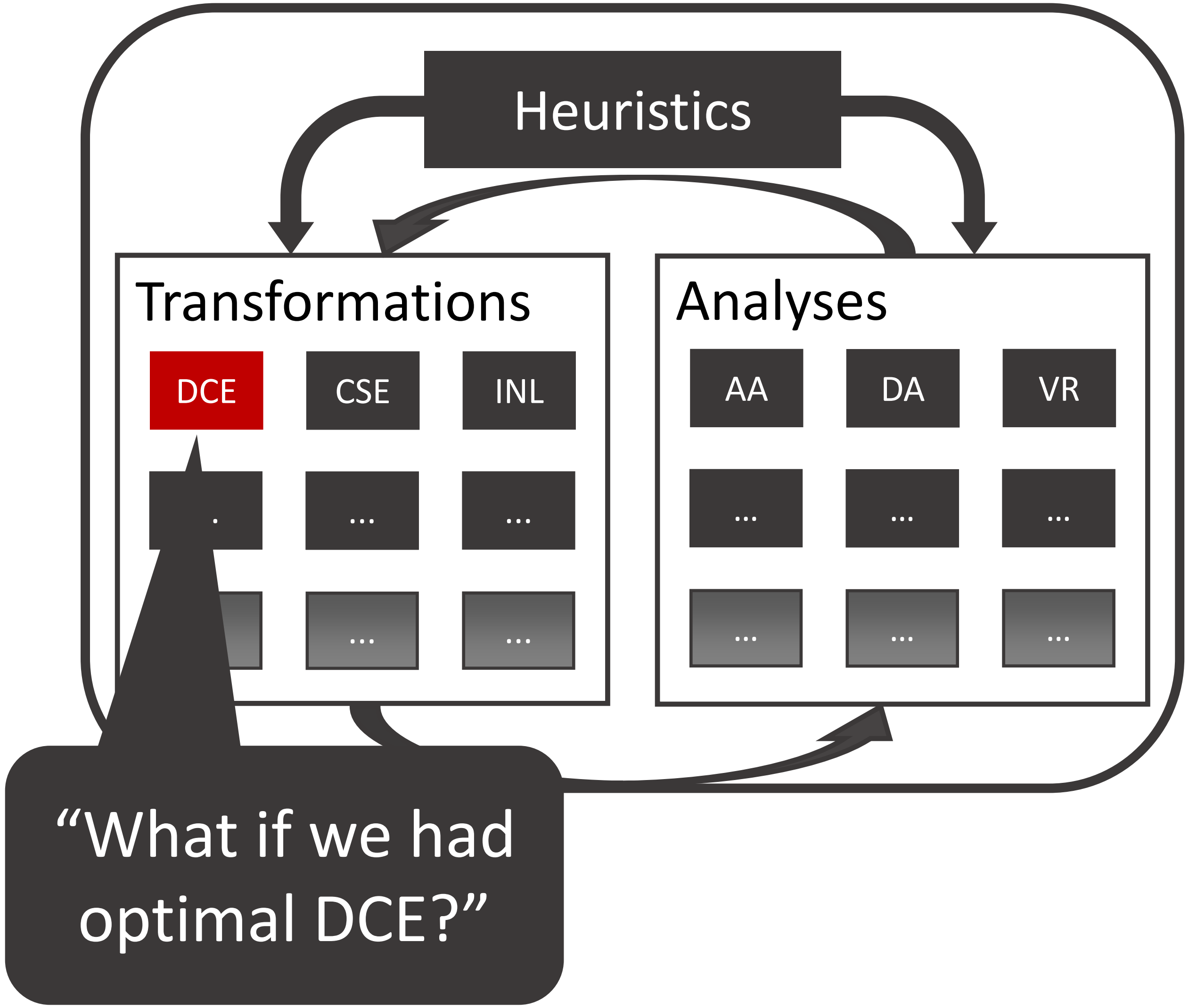
Gap between LLVM and Optimal

Heuristic size overhead



Common inlining choices





Understanding and Exploiting Optimal Function Inlining

Theodoros Theodoridis
theodoros.theodoridis@inf.ethz.ch
ETH Zurich
Switzerland

Tobias Grosser
tobias.grosser@ed.ac.uk
University of Edinburgh
United Kingdom

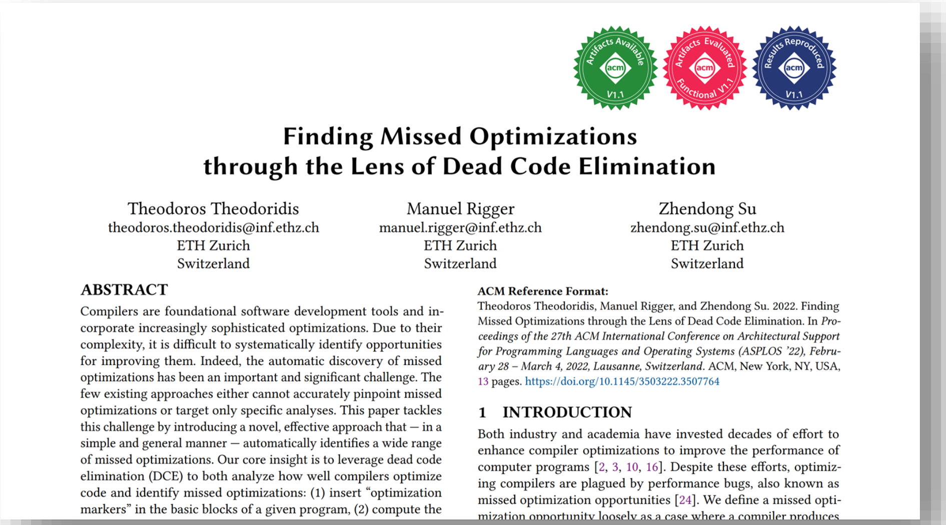
Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT

Inlining is a core transformation in optimizing compilers. It replaces a function call (call site) with the body of the called function (callee). It helps reduce function call overhead and binary size, and more importantly, enables other optimizations. The problem of inlining has been extensively studied, but it is far from being solved; predicting which inlining decisions are beneficial is nontrivial due to interactions with the rest of the compiler pipeline. Previous work has mainly focused on designing heuristics for better inlining decisions and has not investigated *optimal inlining*, i.e., exhaustively finding the optimal inlining decisions. Optimal inlining is necessary for identifying and exploiting missed opportunities and evaluating the state of the art. This paper fills this gap through an extensive empirical analysis of optimal inlining using the SPEC2017 benchmark suite. Our novel formulation drastically reduces the inlining search space size (from 2^{349} down to 2^{25}) and allows us to exhaustively

1 INTRODUCTION

Function inlining (aka *inlining expansion*) is one of the fundamental compiler transformations. Not only does it eliminate function call overhead and potentially shrinks binary size, but it also expands the scope of intra-procedural analyses and optimizations. All of these are enabled by replacing function calls with the callees' bodies. The resulting optimization scope expansion makes inlining a critical transformation. Figure 1 illustrates the importance of inlining.



Finding Missed Optimizations through the Lens of Dead Code Elimination

Theodoros Theodoridis
theodoros.theodoridis@inf.ethz.ch
ETH Zurich
Switzerland

Manuel Rigger
manuel.rigger@inf.ethz.ch
ETH Zurich
Switzerland

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT

Compilers are foundational software development tools and incorporate increasingly sophisticated optimizations. Due to their complexity, it is difficult to systematically identify opportunities for improving them. Indeed, the automatic discovery of missed optimizations has been an important and significant challenge. The few existing approaches either cannot accurately pinpoint missed optimizations or target only specific analyses. This paper tackles this challenge by introducing a novel, effective approach that — in a simple and general manner — automatically identifies a wide range of missed optimizations. Our core insight is to leverage dead code elimination (DCE) to both analyze how well compilers optimize code and identify missed optimizations: (1) insert “optimization markers” in the basic blocks of a given program, (2) compute the

ACM Reference Format:
Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507764

1 INTRODUCTION

Both industry and academia have invested decades of effort to enhance compiler optimizations to improve the performance of computer programs [2, 3, 10, 16]. Despite these efforts, optimizing compilers are plagued by performance bugs, also known as missed optimization opportunities [24]. We define a missed optimization opportunity loosely as a case where a compiler produces

```c
static int a = 0;

int main () {
  if (a != 0) {
    return 1;
  }
  a = 1;
  return 0;
}
```

```asm
main:
  xorl %eax, %eax
  retq
```

COMPILER EXPLORER   Add...   More   Sponsors Backtrace intel Solid Sands   Share   Policies   Other

C source #1

Save/Load   Add new...   Vim   C

```c
static int a = 0;

int main () {
  if (a != 0) {
    return 1;
  }
  a = 1;
  return 0;
}
```

clang 14.0.0

x86-64 clang 14.0.0   -O3

Output...   Filter...   Libraries   Add new...   Add tool...

```asm
main:                                   # @main
        movl    $1, %eax
        cmpb    $0, a(%rip)
        jne     .LBB0_2
        movb    $1, a(%rip)
        xorl    %eax, %eax
.LBB0_2:
        retq
```

Output (0/0)   x86-64 clang 14.0.0   - cached (7549B) ~158 lines filtered

gcc 11.3

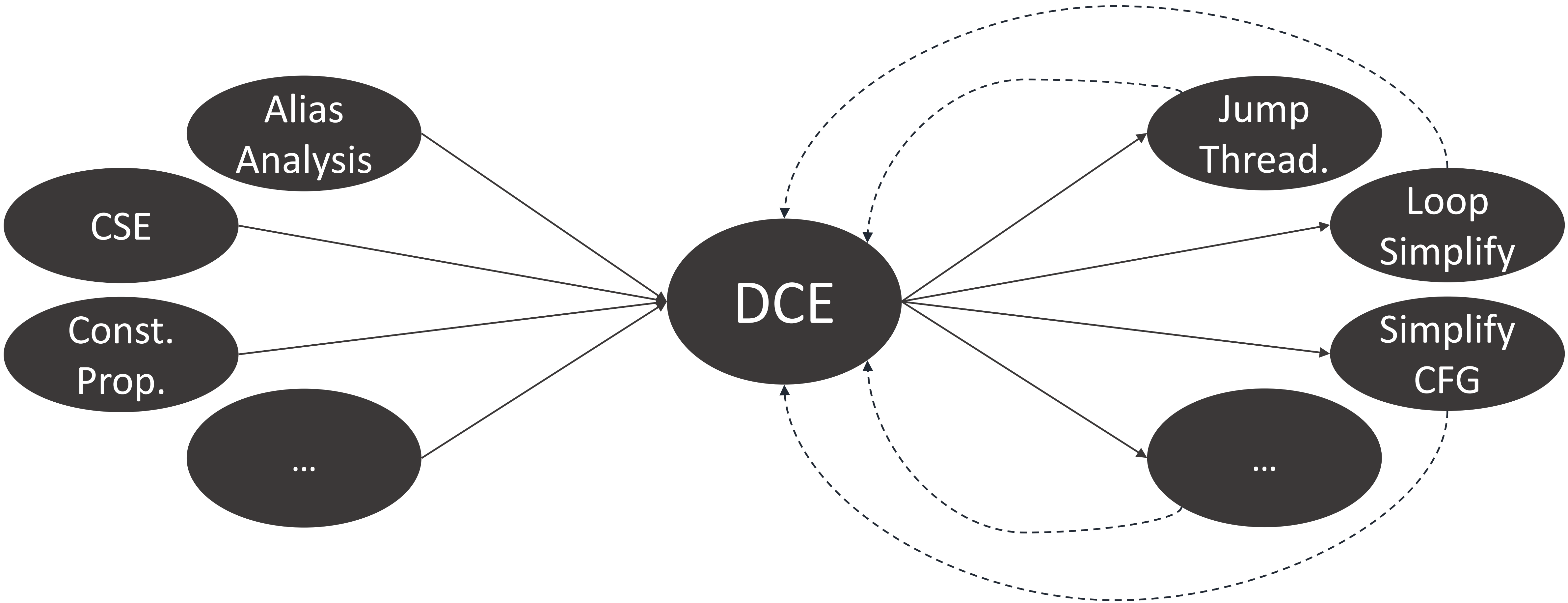
x86-64 gcc 11.3   -O3

Output...   Filter...   Libraries   Add new...   Add tool...

```asm
main:
        movl    a(%rip), %eax
        testl   %eax, %eax
        jne     .L3
        movl    $1, a(%rip)
        ret
.L3:
        movl    $1, %eax
        ret
```


Output (0/0)   x86-64 gcc 11.3   - cached (3047B) ~182 lines filtered

Dead Code Elimination: An Optimization Sink

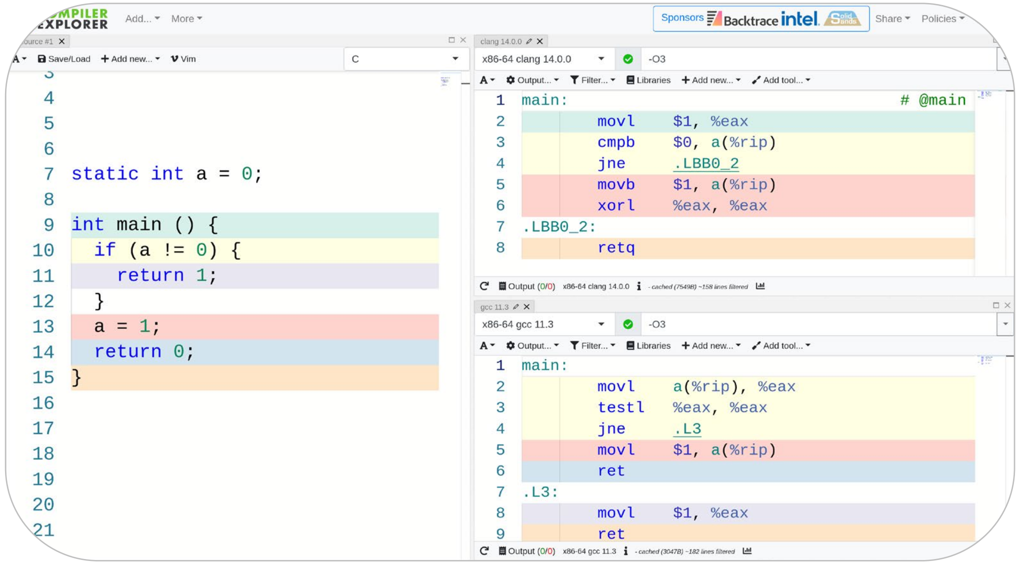


How good are compilers at DCE?

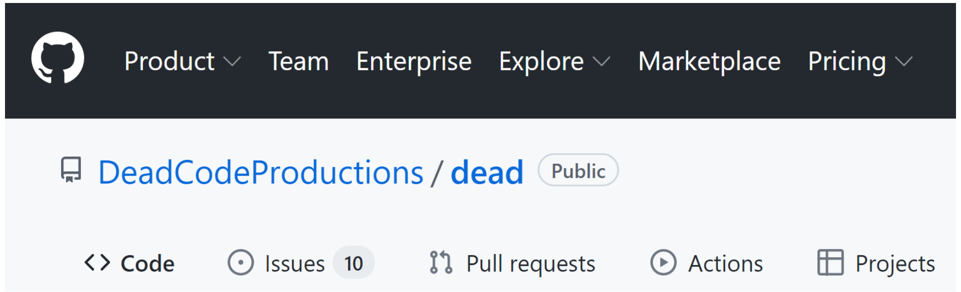
Corpus of 10,000 test programs:

- Generated with Csmith
- 3,109,167 dead blocks

Optimization Level	% of dead blocks that are missed	
	GCC	LLVM
O0	85.2%	83.2%
O1	8.2%	5.2%
Os	6.0%	4.8%
O2	5.7%	4.4%
O3	5.6%	4.3%

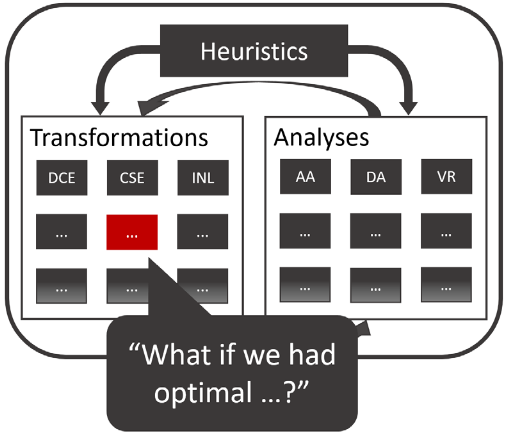


Finding Missed Optimization Opportunities Automatically



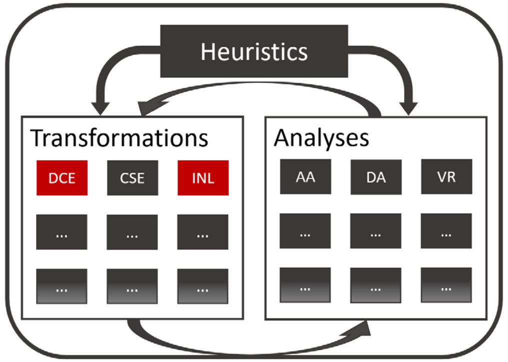
	LLVM	GCC
Reported	47	55
Confirmed	35	46
Fixed	15	15

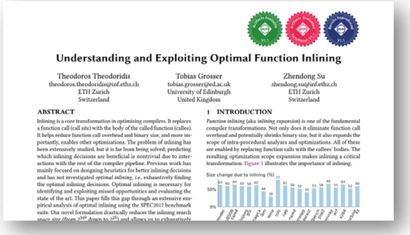
Our Approach

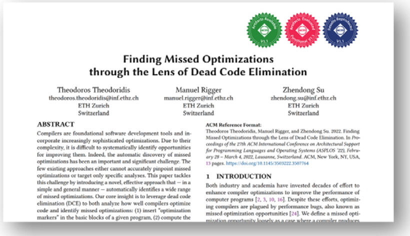


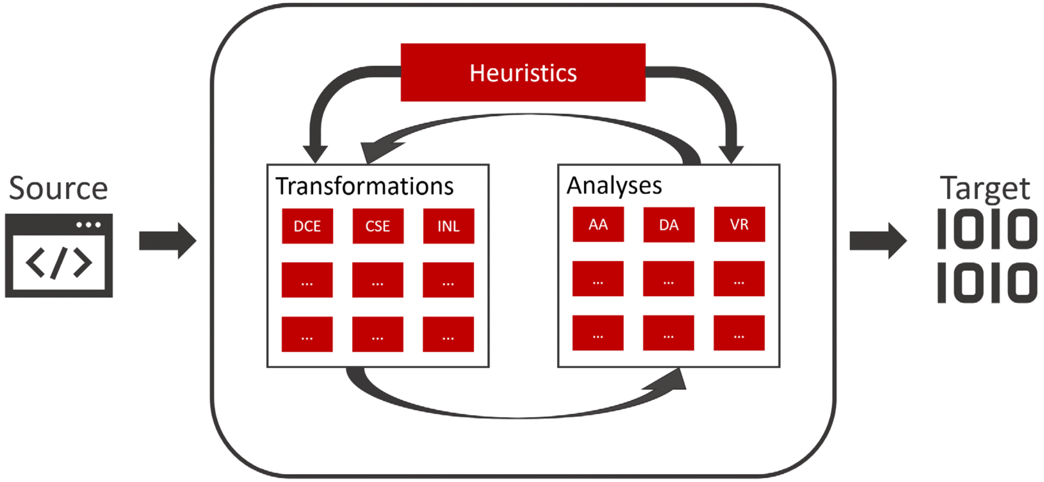
1. We obtain the optimum.

2. We compare with the compiler and find the gap.









Ongoing Work

- Optimal Alias Analysis Information
- Optimal Pass Pipelines
- Learning Heuristics based on Optimal Inlining Choices

Backup Slides

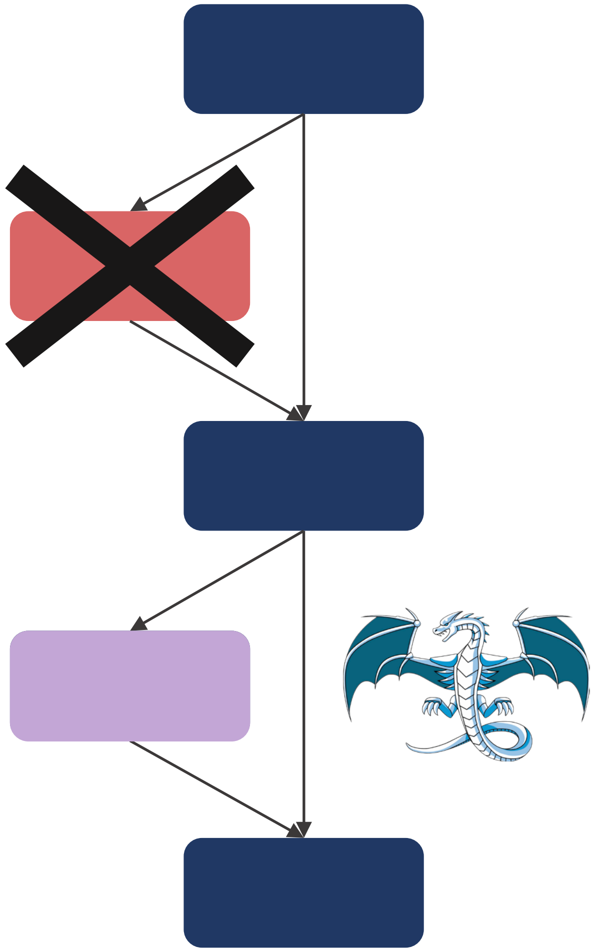
This can be done recursively!

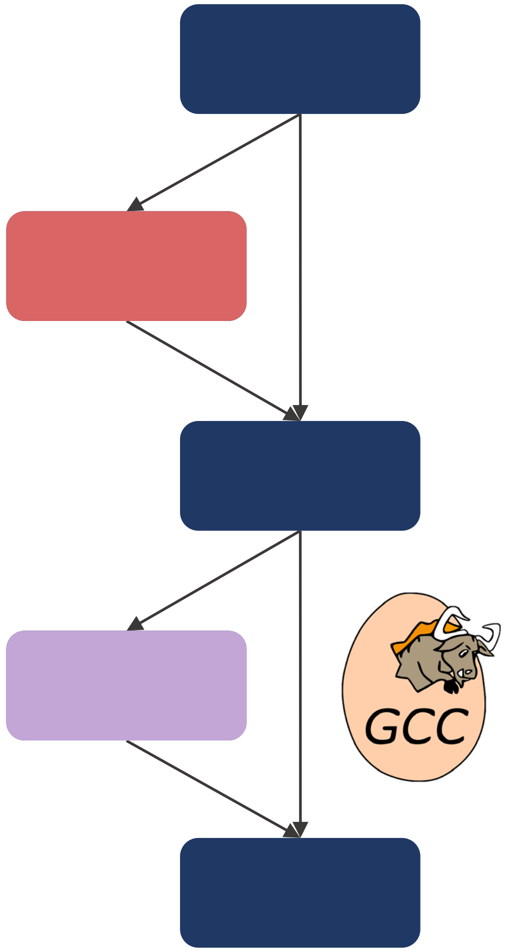
Total:$(2^2 + 2^2 + 1) + 2^4 = 25 < 32$ naïve

Lens of Dead Code Elimination

Different Compilers Eliminate Different Parts

```
int a = 0;
static int b[2] = {0,0}, c = 0;

int main() {
  if (b[a]) {
    return 1;
  }
  if (c) {
    return 2;
  }
  c = 1;
  return 0;
}
```





Missed Dead Code Elimination Detection

```c
int a = 0;
static int b[2] = {0,0}, c = 0;

int main() {
  if (b[a]) {
    return 1;
  }
  if (c) {
    return 2;
  }
  c = 1;
  return 0;
}
```

```
main:
 movl   $2, %eax
 cmpb   $0, c(%rip)
 jne    .LBB0_2
 movb   $1, c(%rip)
 xorl   %eax, %eax
.LBB0_2:
 retq
```

```
main:
 movslq a(%rip), %rdx
 movl   $1, %eax
 movl   b(,%rdx,4),%edx
 testl  %edx, %edx
 jne    .L1
 movl   c(%rip), %eax
 testl  %eax, %eax
 jne    .L4
 movl   $1, c(%rip)
 ret
.L4:
 movl   $2, %eax
.L1:
 ret
```

GCC

Missed Dead Code Elimination: Markers

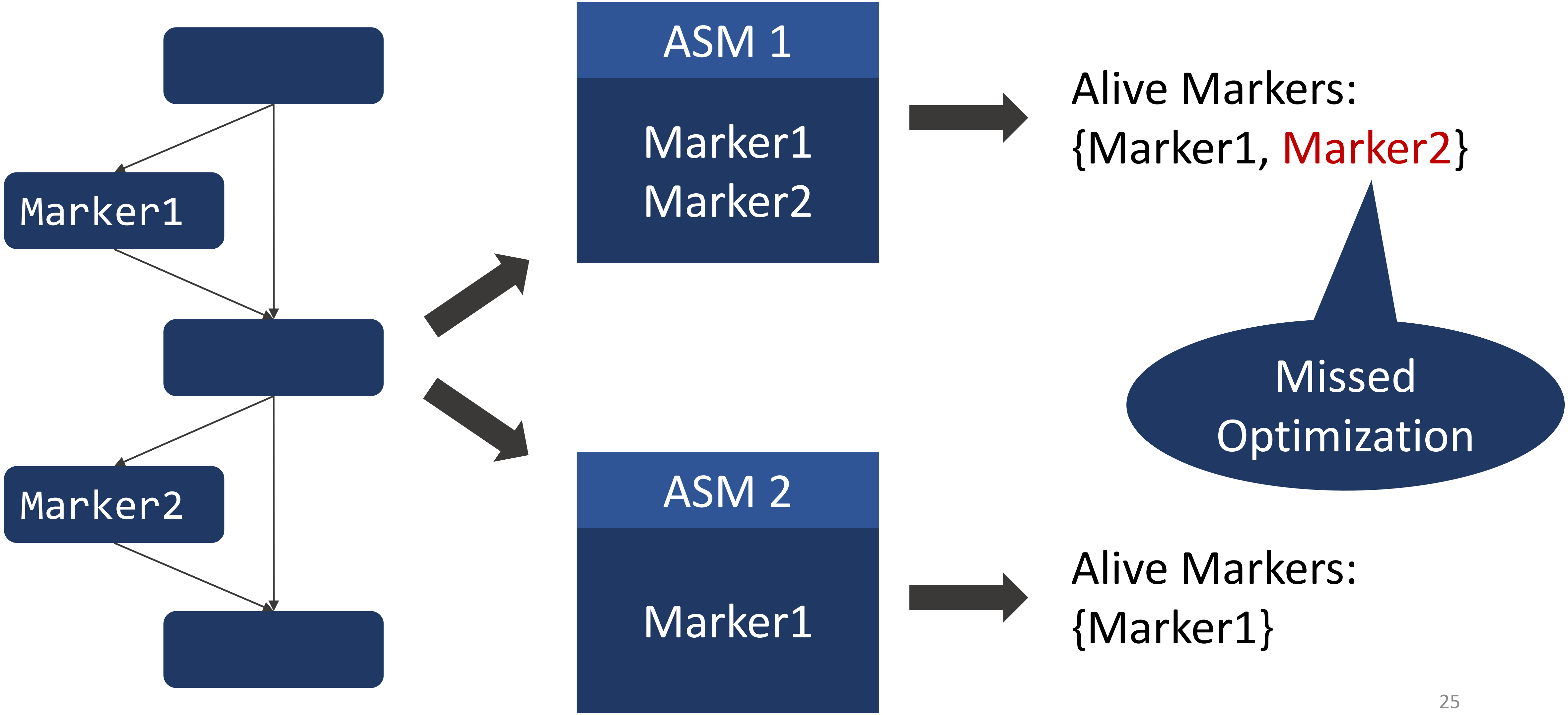
```c
int a = 0;
static int b[2] = {0,0}, c = 0;

int main() {
  if (b[a]) {
    return 1;
  }
  if (c) {
    return 2;
  }
  c = 1;
  return 0;
}
```

```
main:
 pushq    %rax
 cmpb     $1, c(%rip)
 jne      .LBB0_2
 callq    DCEMarker2
 movl     $2, %eax
 popq     %rcx
 retq
.LBB0_2:
 movb     $1, c(%rip)
 xorl     %eax, %eax
 popq     %rcx
 retq
```

```
main:
 subq   $8, %rsp
 movslq a(%rip), %rax
 movl   b(,%rax,4),%eax
 testl  %eax, %eax
 jne    .L7
 …
.L7:
 call   DCEMarker1
 movl   $1, %eax
 jmp    .L1
.L8:
 call   DCEMarker2
 movl   $2, %eax
 jmp    .L1
```

GCC

The Lens of Dead Code Elimination



DCE Examples

```
static int a[2], b, *c[2];

int main() {
  for (b = 0; b < 2; b++) {
    c[b] = &a[1];
  }
  if (!c[0]){
    DCEMarker();
  }
  return 0;
}
```

Pointer data vectorized as `unsigned int`

Vectorized at –O3

c[0] points to a non-zero address

GCC -O1 ✓

GCC -O3 ✗

```c
static long a = 78240;
static int b, d;
static short e;
static short c(short f, short h) {
  return h == 0 ||
   (f && h == 1) ? 0 : f % h; }
int main() {
  short g = a;
  for (b = 0; b < 1; b++) {
    e = a;
    d = c((e == a) ^ g, a);
  }

  if (d) {
    DCEMarker();
    for (; a; a++);
  }
}
```

Modulo on constant ranges: [X,X+1) % [X,X+1) not simplified

LLVM 13 -O1

LLVM 13 -O3

```
static int b = -1, e = 1;
static short c = 0, d = 0;
short a(unsigned short f, int g) {
  return f >> g;
}

int main() {
  c++;
  d  = a(4294967295 + (c > 0),1);
  e ^= (short)(d * 3) /(unsigned)b;
  if (!e)
    DCEMarker();
}
```

Regression on shift peephole optimization

e != 0

LLVM 13 -O3 ✓

LLVM dev -O3 ✗

[SimplifyCFG] don't sink common insts too soon (PR34603)

This should solve:
https://bugs.llvm.org/show_bug.cgi?id=34603
...by preventing SimplifyCFG from altering redundant instructions before early-cse has a chance to run.