systems-rg.github.io

Systems Reading Group 2022

Understanding and Exploiting Optimal Function Inlining

Theodoros Theodoridis, Tobias Grosser, Zhendong Su

Discussion Lead : Sorav Bansal

Compiler Optimization : Why bother?

- Proebsting's Law: Compiler Advances Double Computing Power Every Twenty Years

- This paper will show improvements of 1-15%

For

- Quoting venturebeat: … expects "an explosion" in the importance and adoption of tools like PyTorch's JIT ***compiler*** and neural network hardware accelerators like Glow

- Huge compiler teams at hardware companies: Intel, Qualcomm, AMD, nVIDIA, Microsoft, Google, …

Against

Compiler Optimization : Why bother?

Compiler Optimization Research Will Drive Innovations in Computer Systems for the next 50 years

Sorav’s Law, stated in 2022 ;)

Which Compiler Optimizations Matter?

Which Compiler Optimizations Matter Most?

- Inlining
- Vectorization (SIMD)
- Scheduling for Parallelization
- Scheduling for Locality
- Register Allocation
- Loop Invariant Code Motion
- Common Subexpression Elimination
- Dead Code Elimination
- Constant Propagation
- Peephole Optimizations…

Typical Improvement

X

Typical Frequency of Occurrence

Which Compiler Optimizations Matter Most?

Traditional

- Inlining
- Vectorization (SIMD)
- Scheduling for Parallelization
- Scheduling for Locality
- Register Allocation
- Loop Invariant Code Motion
- Common Subexpression Elimination
- Dead Code Elimination
- Constant Propagation
- Peephole Optimizations…

Modern

- Inlining of Operators
- Auto-Distribution
- Auto-Parallelization
- Automatic Data Placement
- Automatic Cache Management
- Memoization
- Common Subexpression Elimination
- Dead Code Elimination
- Constant Propagation
- Peephole Optimizations…

Inlining : One of the Most Consequential Transformations

```
class Foo {
private:
  int m = 0;
public:
  int get_m() const { return m; }
  void inc_m() { m++; }
  …
}
```

```
Foo foo;

for (; foo.get_m() < n;
       foo.inc_m())
{
   …
}
```

Inlining is often a prerequisite for transformations like loop vectorization

Inlining is More Consequential in Higher Level Languages

- Utility Functions (C)
- Getters and Setters (C++, etc.)
- Lambdas (C++, etc.)
- Custom Operators (e.g., Map/Reduce) that accept arbitrary functions
- Stream Processing Languages
    - Stream operators composed in sequence can be inlined into optimized sequential code
    - Examples of a follow-up transformation: Operator Scheduling, Parallelization/Vectorization
- Neural Network Languages like Tensorflow
    - Inline Neural Network Operators composed in sequence
    - Examples of a follow-up transformation: Polyhedral transformations, Parallel/Vectorization

nadav.amit.zone/linux/2018/10/10/newline.html

Nadav Amit

Publications Contact Service

Up to 10% performance degradation (unpredictable)

Conclusion: Inlining heuristics are fragile

How new-lines affect the Linux kernel performance

Oct 10, 2018

The Linux kernel strives to be fast and efficient. As it is written mostly in C, it can mostly control how the generated machine code looks. Nevertheless, as the kernel code is compiled into machine code, the compiler optimizes the generated code to improve its performance. The kernel code, however, employs uncommon coding techniques, which can fail code optimizations. In this blog-post, I would share my experience in analyzing the reasons for poor code inlining of the kernel code. Although the performance improvement are not significant in most cases, understanding these issues are valuable in preventing them from becoming larger. New-lines, as promised, will be one of the reasons, though not the only one.

New lines in inline assembly

One fine day, I encountered a strange phenomenon: minor changes I performed in the Linux source code, caused small but noticeable performance degradation. As I expected these changes to actually improve performance, I decided to disassemble the functions which I changed. To my surprise, I realized that my change caused functions that were previously inlined, not to be inlined anymore. The decision not to inline these functions seem dubious as they were short.

This paper…

- Understanding Optimal Function Inlining

- Exploiting it

Focus on Code Size (-Os)

- More Inlining → More Optimization Opportunities (e.g., Dead Code Elim.)
- More Inlining → More Code Bloat

Inlining Example

```c
int bar(int a) {
   return a + a;
}

int foo(int n) {
   for (int i = 0; i < n; ++i)
   {
      if (bar(i) == i)
         return 0;
   }
   return 1;
}
```

Listing 1: Source Code

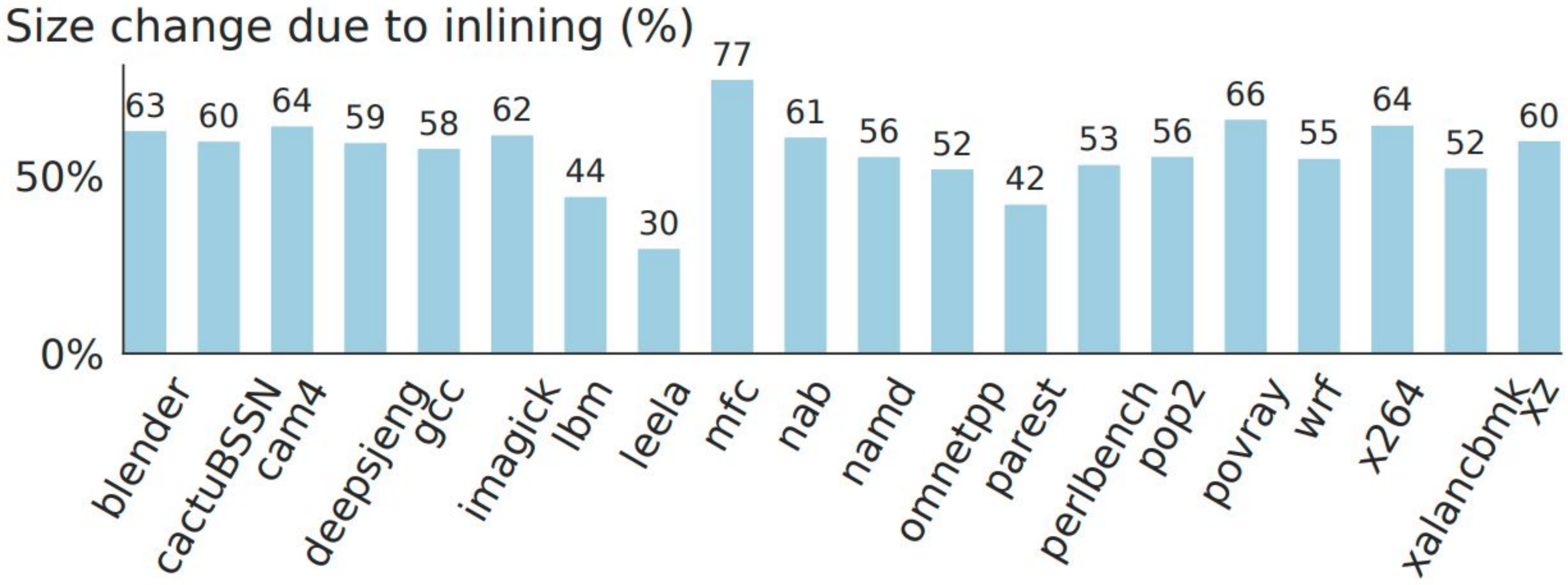
```
foo:
  xorl   %eax, %eax
  testl  %edi, %edi
  setle  %al
  retq
```

Listing 2: foo inlined

```
foo:
  pushq %rbp
  pushq %r14
  pushq %rbx
  movl  $1, %r14d
  testl %edi, %edi
  jle   .LBB1_5
  movl  %edi, %ebp
  xorl  %ebx, %ebx
.LBB1_3:
  movl  %ebx, %edi
  callq bar
  cmpl  %eax, %ebx
  je    .LBB1_4
  addl  $1, %ebx
  cmpl  %ebx, %ebp
  jne   .LBB1_3
  jmp   .LBB1_5
.LBB1_4:
  xorl  %r14d, %r14d
.LBB1_5:
  movl  %r14d, %eax
  popq  %rbx
  popq  %r14
  popq  %rbp
  retq
```

Listing 3: foo not inlined

Understanding Optimal Function Inlining (LLVM -Os)



SPEC CPU 2017

Identifying the Optimal Inlining Configuration is NP-Hard

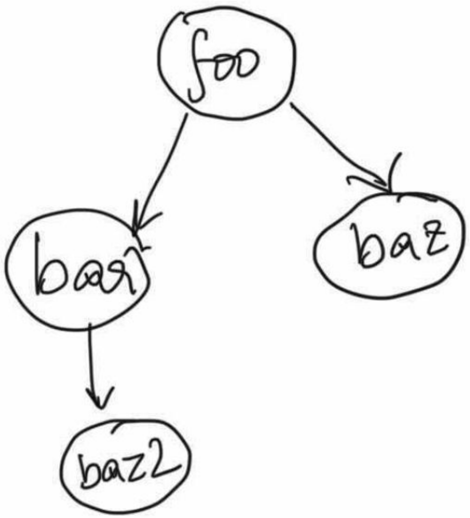
- State of the Practice: Heuristics (e.g., size of callee)

- Research Ideas: “Inlining Trials” during Compiler Optimization

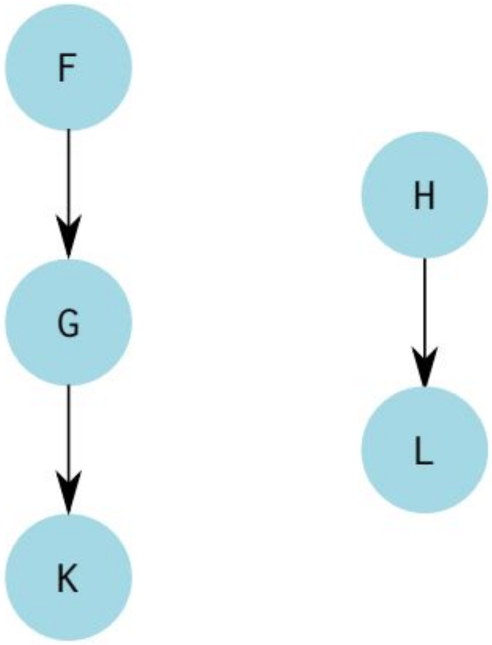
- Idea: “Put Inlining Trials on Steroids”, but during an offline phase that can take tens of hours on hundreds of CPUs
    - What is the best algorithm to identify the optimal inlining configuration (even though exponential time)
    - What insight does it provide? Can the insights be used to identify a *fully parallel* algorithm that

Inlining Search Space

- Identify Inlinable Functions (e.g., no recursion)
- Construct a call graph (e.g., if foo() calls bar() and baz(), and bar() calls baz2()) .   Label each edge as “inline” or “no-inline” (exponential space)
- Naive algorithm: $O(2^{|E|})$
- This assumes “coupled inlining decisions”
    - If (bar→baz) is inlined, then it would be inlined everywhere
        - e.g., (foobar→ baz) and (foo2bar→baz) will be inlined

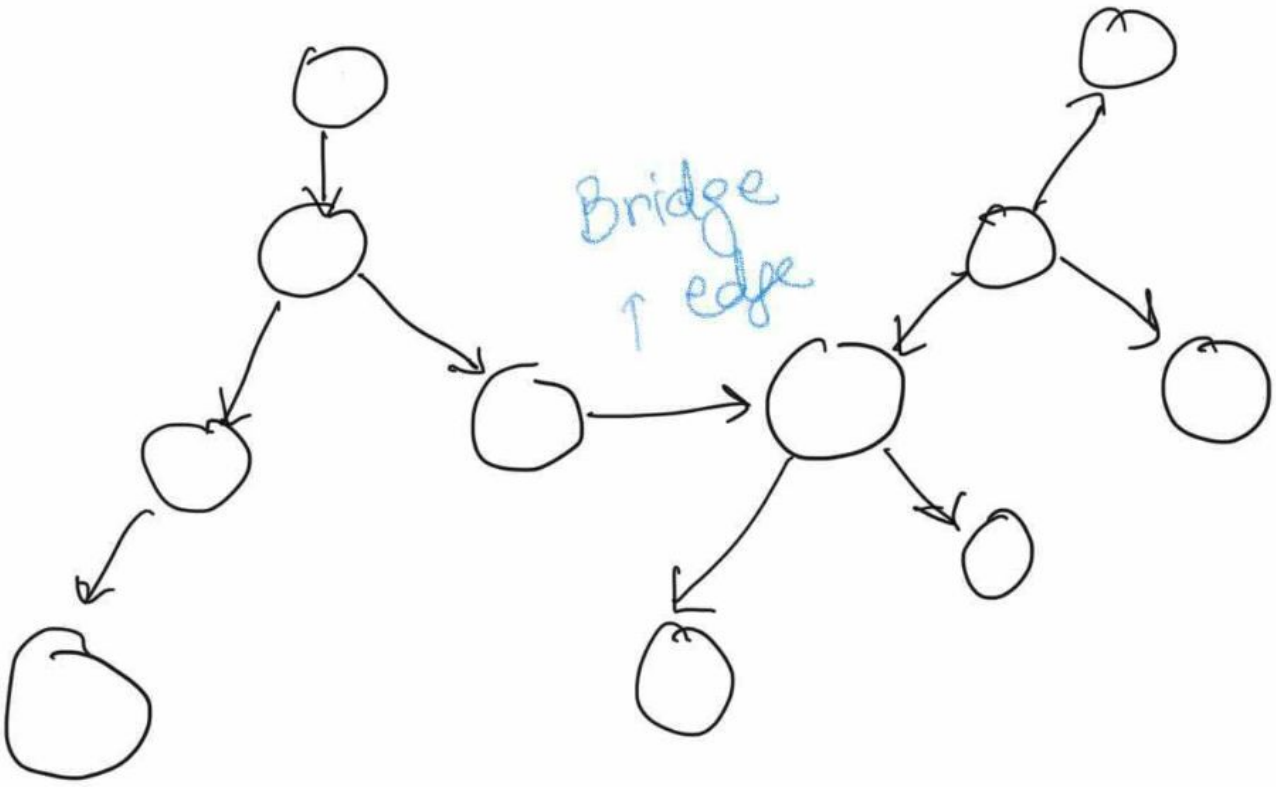


Improvement: Separate into CCs (Connected Components)

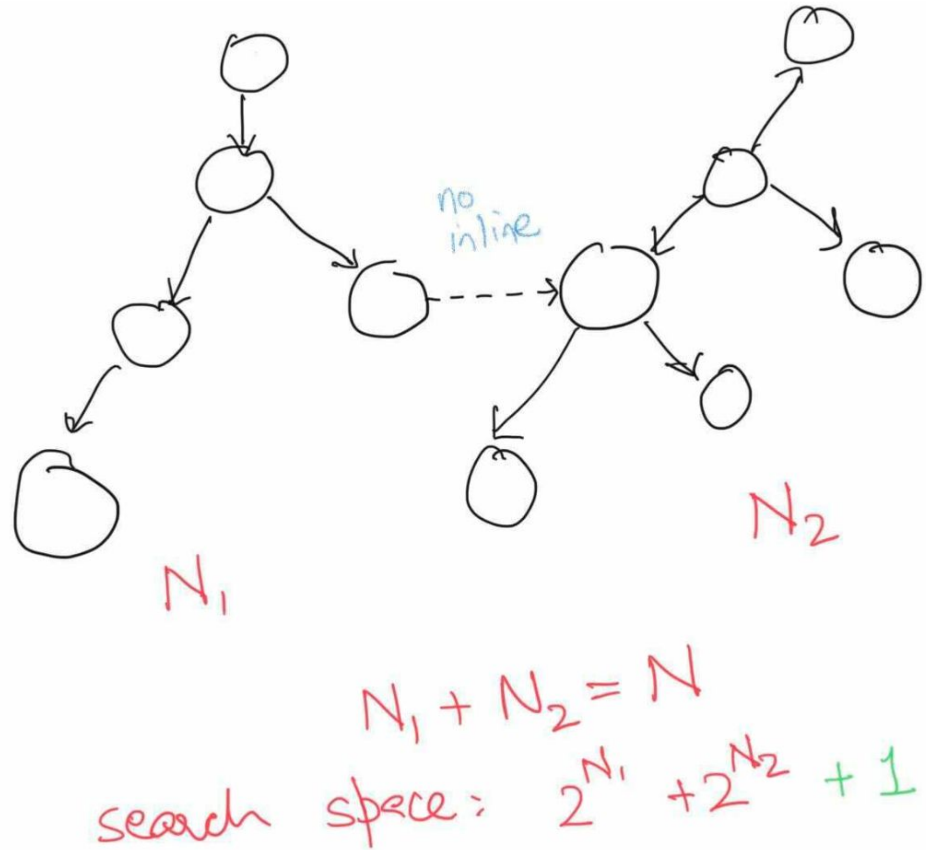


Component Inlining Configurations		
F $\to$ G	G $\to$ K	H $\to$ L
no-inline	*no-inline*	*no-inline*
no-inline	*inline*	*inline*
inline	*no-inline*	
inline	*inline*	

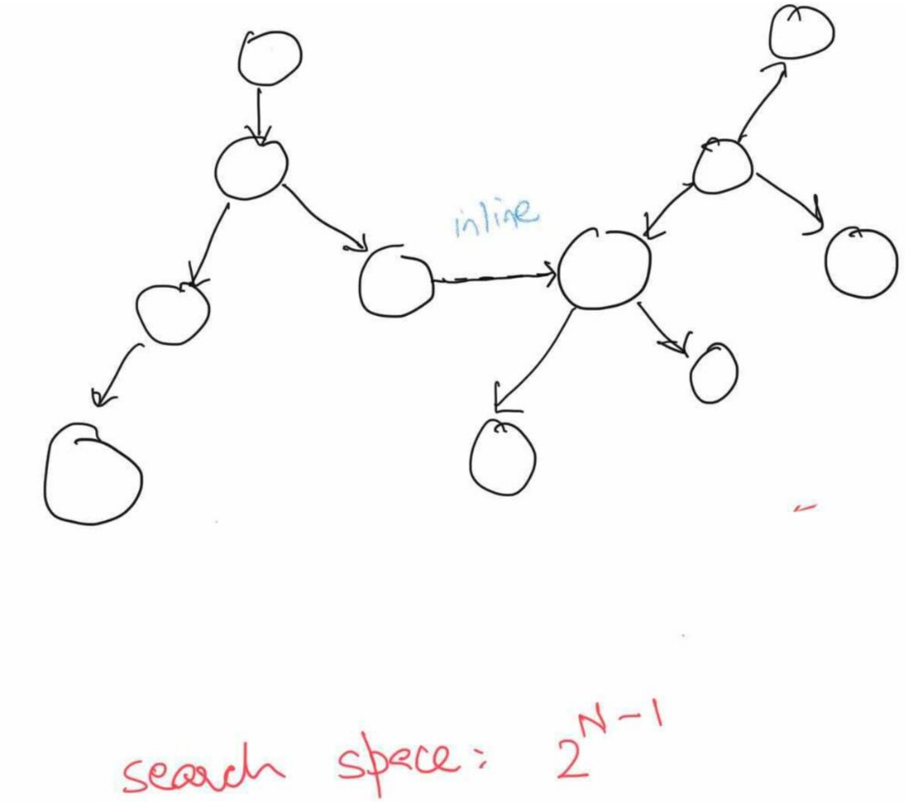
Recursively Partitioned Search Space

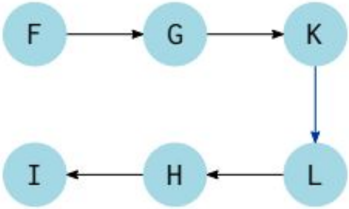


Recursively Partitioned Search Space

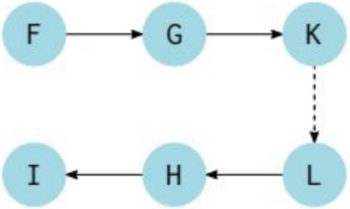


Recursively Partitioned Search Space

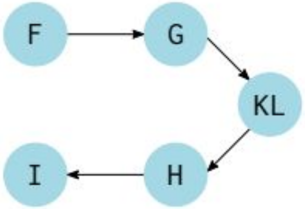




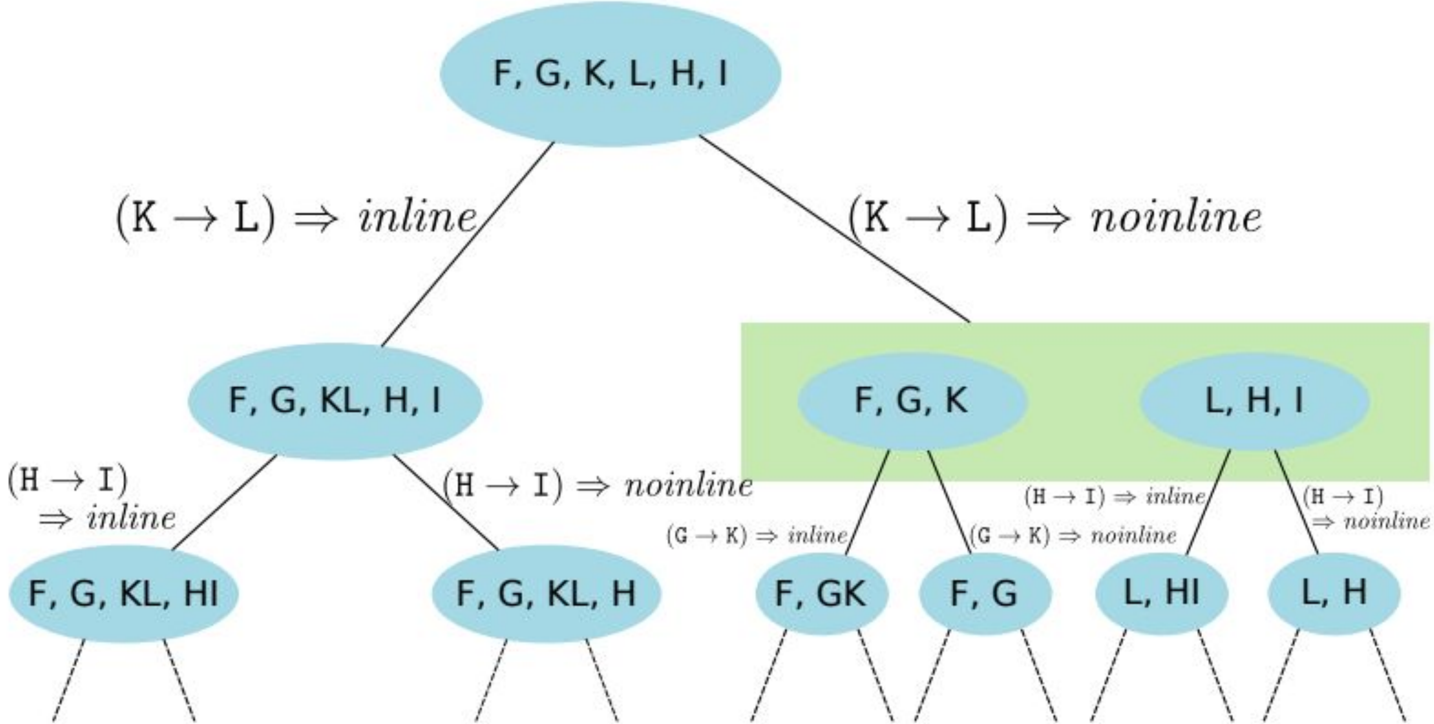
(a) original



(b) K $\rightarrow$ L **not inlined**



(c) K $\rightarrow$ L **inlined**



Choice of Bridge Edge

- Determines the size of the Search Space
- Heuristically choose the bridge edge to try and divide the call graph into many independent components of roughly equal size
    - Edge incident to *least eccentric vertex*
        - Vertex with least maximum distance from any other vertex

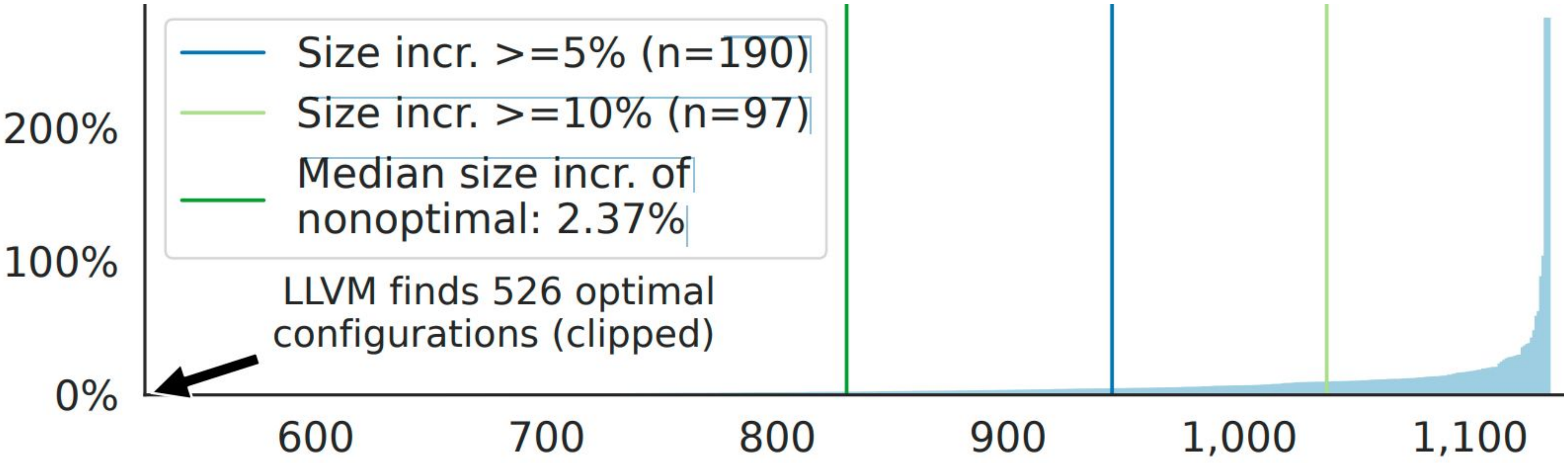
Search Space Reduction

Search Space	Per file size percentiles (log 2)				Geometric Mean
	Median	75th	95th	Max	
naïve	8	18	38	349	7.57
recursive	6.2	10.9	17.4	19.9	5.42

approximately $2^{349} \rightarrow 2^{25.2}$

Comparison with LLVM Heuristics

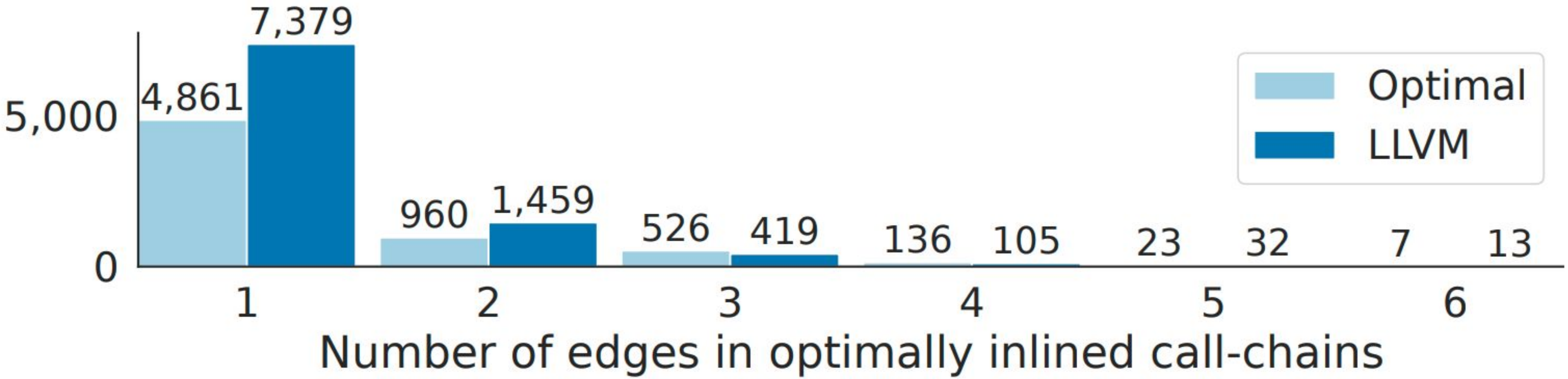
Max number of inlinable calls=1135; Max search space=2^{18}



In 23.7% cases, LLVM's heuristic is inlining too aggressively

Length of Inlined Call Chains

Max number of inlinable calls=1135; Max search space=2^{18}



Observation

- Optimal configurations have small length inlined call chains

- Redefine the search space : consider only those cases where the optimal is
  - Either no-inline for all edges
  - Or has inlined call-chains of less than 1

- Identify an efficient embarrassingly-parallel algorithm that can identify the optimal in this redefined search space; and see how it works for other cases (outside this redefined search space)

Autotuner Algorithm

Start with a call graph, say CG

For each edge (in parallel)

- Inline that edge in CG, and perform the rest of the compiler transformations
- See if the inlining of the edge reduced the code size. If yes, mark that edge as “inline” in the final solution

Suboptimal if the inlining of either “A” or “B” *reduces* (increases) code size, but inlining of both “A” and “B” *increases* (reduces) code size

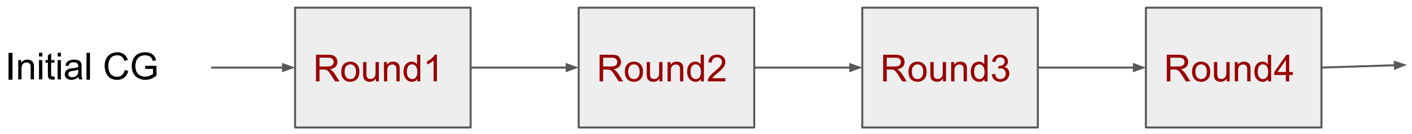
Autotuner Algorithm

Start with a call graph, say CG

One round

For each edge (in parallel)

- Inline that edge in CG, and perform the rest of the compiler transformations
- See if the inlining of the edge reduced the code size. If yes, mark that edge as “inline” in the final solution



Single Round Results (starting from clean slate)

Figure 10: Autotuning (clean slate) versus LLVM -Os on SPEC2017. Out of the 20 benchmarks: 14 shrink in size, 1 remains unchanged, and 5 inflate. The median relative size is 97.95%. The largest benchmark size reduction is 27.6% (`mfc`).

Single Round Results (starting from “llvm -Os” output)

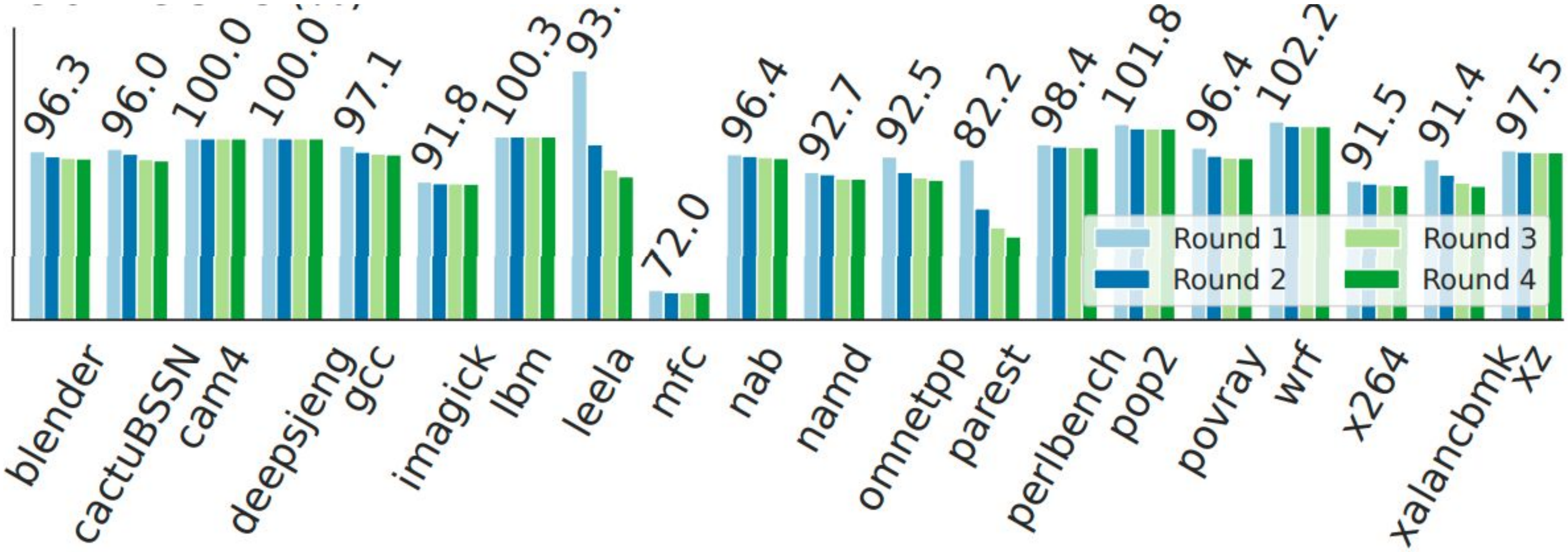
Figure 12: LLVM-initialized autotuning versus LLVM -Os on SPEC2017. Out of the 20 benchmarks: 19 shrink in size, 1 remains unchanged. The median relative size is 97.6%. The largest benchmark size reduction is 21% (`mfc`).

Starting from clean-slate is often better than starting from "llvm -Os"

Table 3: Benchmarks faring worse with LLVM-initialization.

Benchmark	Autotuned relative size vs LLVM -Os	
	Clean slate	LLVM-initialized
imagick	92.1%	96.3%
mfc	72.4%	79%
nab	97.1%	98.8%
nambd	93.9%	95.2%
perlbench	98.9%	99.6%
x264	92.3%	94.1%
xz	97.8%	97.9%

Multiple Rounds



(b) Clean slate, per round medians: 97.95% , 97.02% , 96.46% , 96.38%

Example of the Effect of Multiple Rounds

	LLVM	Round 1	Round 2	Round 3	Round 4
# inlined	114	109	112	107	109
# non inlined	35	40	37	42	40
Rel. Size	100%	71.6%	41.2%	41.4%	35.8%

Table 4: `523.xalancbmk/XalanBitmap.cpp` inlining changes across rounds of LLVM-initialized autotuning.

More Results

- LLVM
    - 84.74% of “LLVM -Os”
    - Took 44-53 hours of auto-tuning
- SQLite
    - X86 backend : 89.7% of “LLVM -Os”
    - WASM backend: 98.74% of “LLVM -Os”. Why such less improvement for WASM?
- Mean slowdowns of 3.6% on SPEC benchmarks

Take-Aways

- Heuristic Recursive Partitioning is interesting and effective
- A Gold-Standard for Inlining Research
- Can be used for “training ML models”
- Exhaustive Search for Performance
  - Let’s not be afraid of Exponentials anymore