# Function Order Optimizations for Mobile Apps

Ellis Hoag

# Optimizing Function Layout for Mobile Applications

ELLIS HOAG, Meta Platforms Inc., USA
KYUNGWOO LEE, Meta Platforms Inc., USA
JULIÁN MESTRE, Meta Platforms Inc., USA and University of Sydney, Australia
SERGEY PUPYREV, Meta Platforms Inc., USA

Function layout, also referred to as function reordering or function placement, is one of the most effective profile-guided compiler optimizations. By reordering functions in a binary, compilers are able to greatly improve the performance of large-scale applications or reduce the compressed size of mobile applications. Although the technique has been studied in the context of large-scale binaries, no recent study has investigated the impact of function layout on mobile applications.

In this paper we develop the first principled solution for optimizing function layouts in the mobile space. To this end, we identify two important optimization goals, the compressed code size and the cold start-up time of a mobile application. Then we propose a formal model for the layout problem, whose objective closely matches the goals. Our novel algorithm to optimize the layout is inspired by the classic balanced graph partitioning problem. We carefully engineer and implement the algorithm in an open source compiler, LLVM. An extensive evaluation of the new method on large commercial mobile applications indicates up to 2% compressed size reduction and up to 3% start-up time improvement on top of the state-of-the-art approach.

CCS Concepts: • **Software and its engineering** → *Compilers*; • **Theory of computation** → **Graph algorithms analysis**.

Additional Key Words and Phrases: profile-guided optimizations, code layout, function reordering, code-size reduction, graph algorithms

https://arxiv.org/abs/2211.09285

# Optimization Goals

- Mobile apps must launch quickly
  - Fewer `__text` page faults → faster startup
  - Smaller `__text` size → fewer `__text` page faults
    - Compile with **`-Oz`** (optimize for size)
- Mobile apps must be small
  - `.ipa` size (compressed)
  - `.app` size (uncompressed)
    - ~80% of size is executables

# Profile Guided Optimization (PGO)

- LLVM IRPGO
  - `-fprofile-generate`
  - `-fprofile-use`
- Guides inlining decisions → larger binaries
  - `-disable-preinline`
  - `-pgso=false`
    - (profile guided **size** optimization)

# Temporal Profiling

- Goal: Improve startup performance with an orderfile
- `-pgo-temporal-instrumentation`
- Profiles *function timestamps*
- Available since LLVM 17.x
  - https://discourse.llvm.org/t/rfc-temporal-profiling-extension-for-irpgo/68068
  - https://reviews.llvm.org/D147287
- Compatible with Lightweight Instrumentation
  - https://discourse.llvm.org/t/instrprofiling-lightweight-instrumentation/59113
  - https://youtu.be/vFWwJrOiVMM

# Temporal Profiling

```
int global_timestamp = 1;

void foo() {
  if (*timestamp == 0)
    *timestamp = global_timestamp++;
  ...
}
```

```
default-1.profraw
foo:
  timestamp: 1
  counts: ...
goo:
  timestamp: 2
  counts: ...
bar:
  timestamp: 3
  counts: ...
```

```
default-2.profraw
foo:
  timestamp: 1
  counts: ...
goo:
  timestamp: 0
  counts: ...
bar:
  timestamp: 3
  counts: ...
```

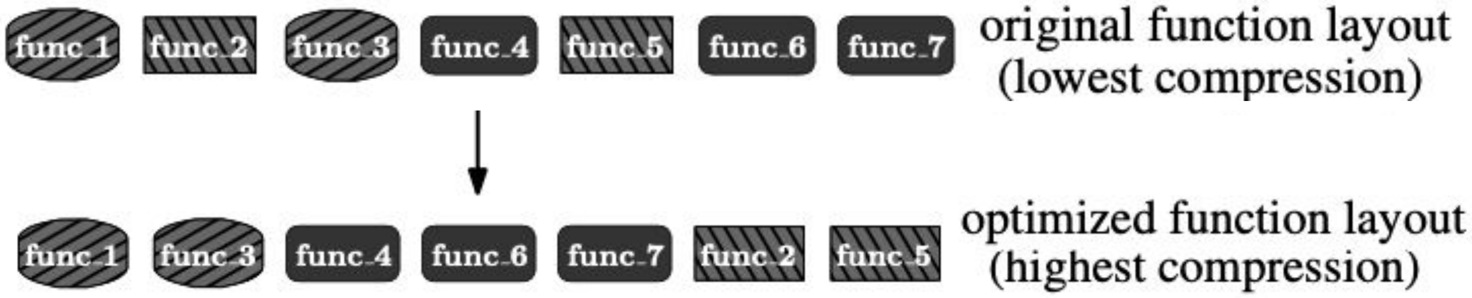
```
Trace 1
foo, goo, bar
```

```
Trace 2
foo, bar
```

```
default.profdata
Counts: ...
Trace 1:
      foo, goo, bar
Trace 2:
      foo, bar
```

# Compressed Size

- Compressed size depends on # distinct sequences in a 64KB sliding window
  - Co-locating `similar` functions can improve the compression ratio



# Balanced Partitioning

- Orders functions to minimize some objective
  - Use function traces to minimize page faults
  - Use function contents to minimize compressed size
- Performant
  - Can order 1M functions in 20 seconds
- LLVM 17.x
  - https://reviews.llvm.org/D147812

# Balanced Partitioning

### Original Functions

```
f1:
  add x0, x0, 1
  br x0
```

```
f2:
  mul x0, x0, 2
  add x0, x0, 1
  ret
```

```
f3:
  add x0, x0, 1
  mul x0, x0, 2
  br x0
```

```
f4:
  mul x0, x0, 2
  ret
```

→

### Stable Hash

```
f1:
 0x10101010(u1)
 0x20202020(u2)
```

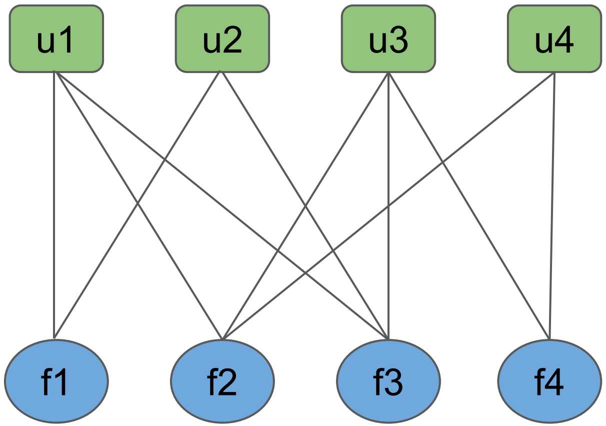
```
f2:
 0x30303030(u3)
 0x10101010(u1)
 0x40404040(u4)
```

```
f3:
 0x10101010(u1)
 0x30303030(u3)
 0x20202020(u2)
```

```
f4:
 0x30303030(u3)
 0x40404040(u4)
```

→

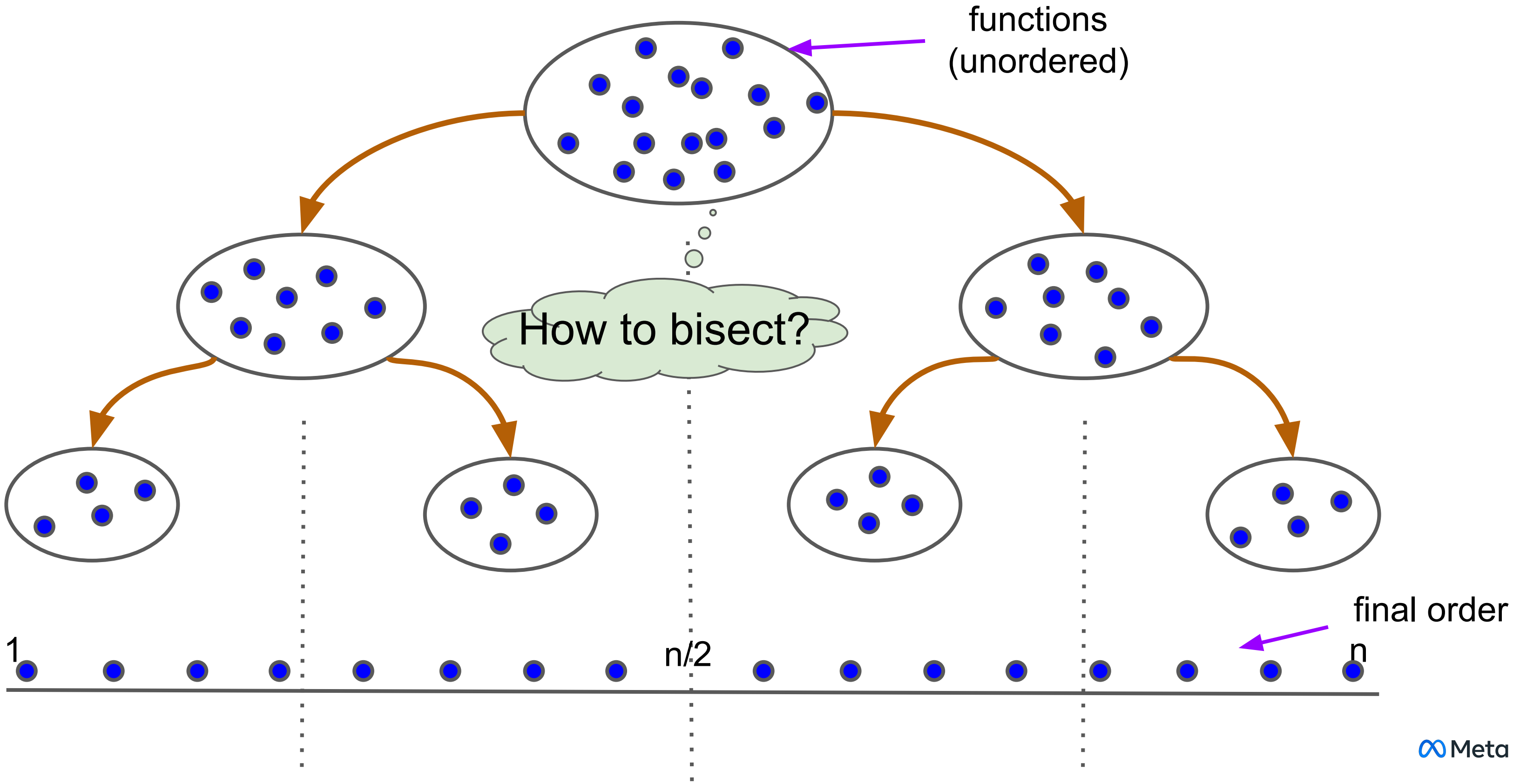
### Utility Vertices




### Functions

**Goal: Place functions sharing utilities nearby!**

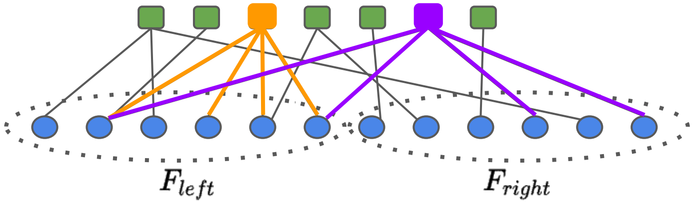
# Recursive Balanced Partitioning



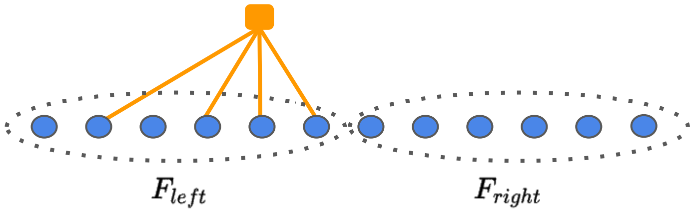
# Balanced Partitioning

utilities:

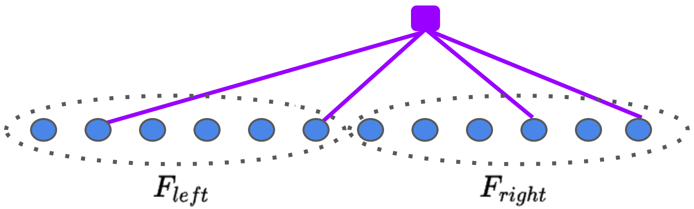
functions:



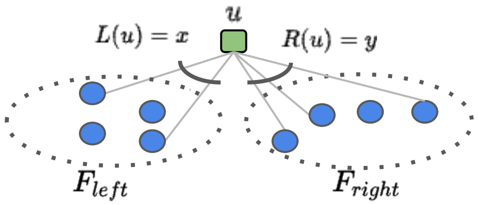

utility: ✅




utility: ❌




**Objective: Minimize # utilities spanning two buckets by swapping functions**




$$\sum_{u \in U} cost(L(u), R(u)) = \sum_{u \in U} -(x \cdot log(x+1) + y \cdot log(y+1))$$

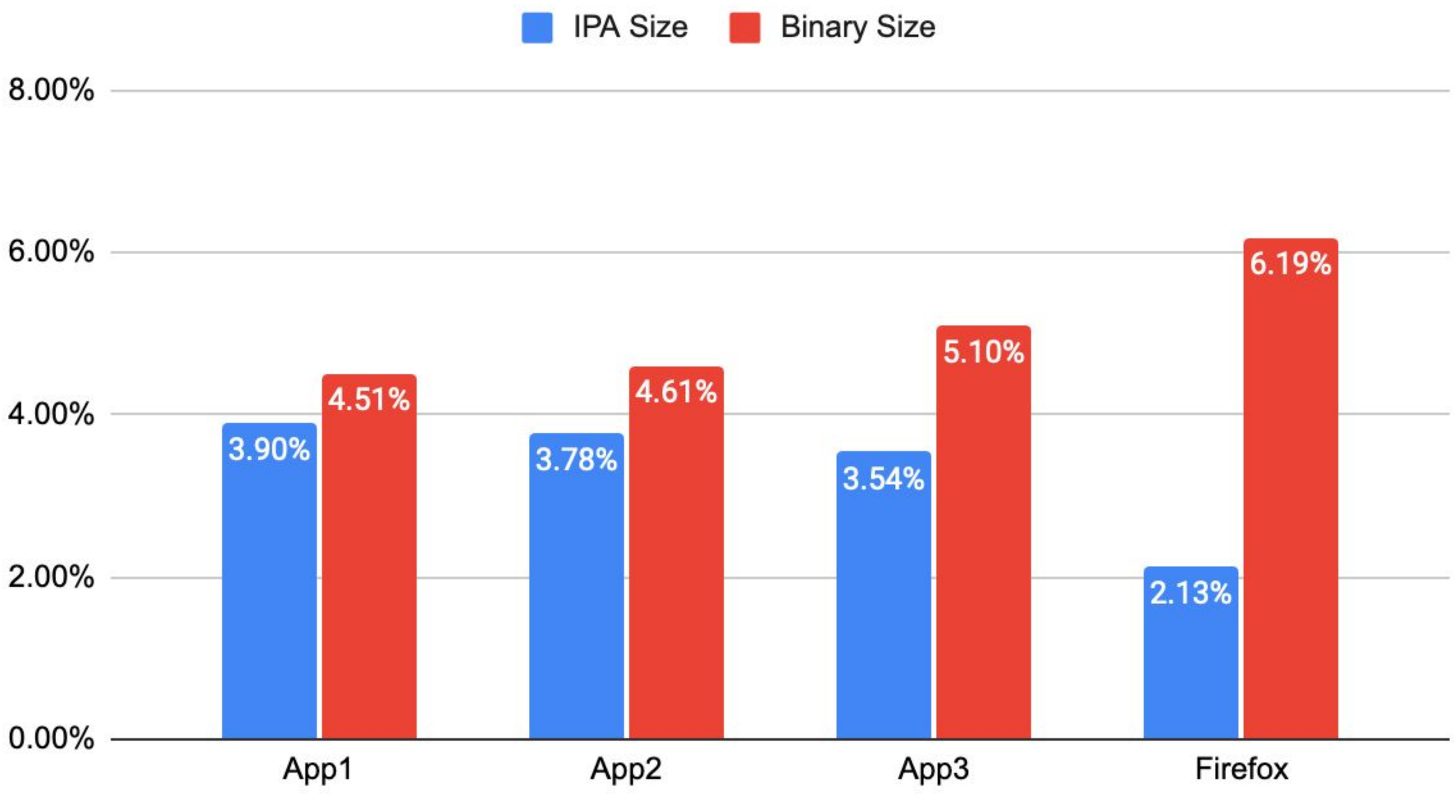
# When to Order Functions?

- Before building?
  - `$ llvm-profdata order default.profdata -o a.orderfile`
  - `$ clang -Wl,-order_file,a.orderfile ...`
  - ✅ Function traces for startup
  - ❌ No stable hashes for compression
- Link time
  - ✅ Function traces for startup
  - ✅ Stable hashes for compression
  - ✅ Safe optimization

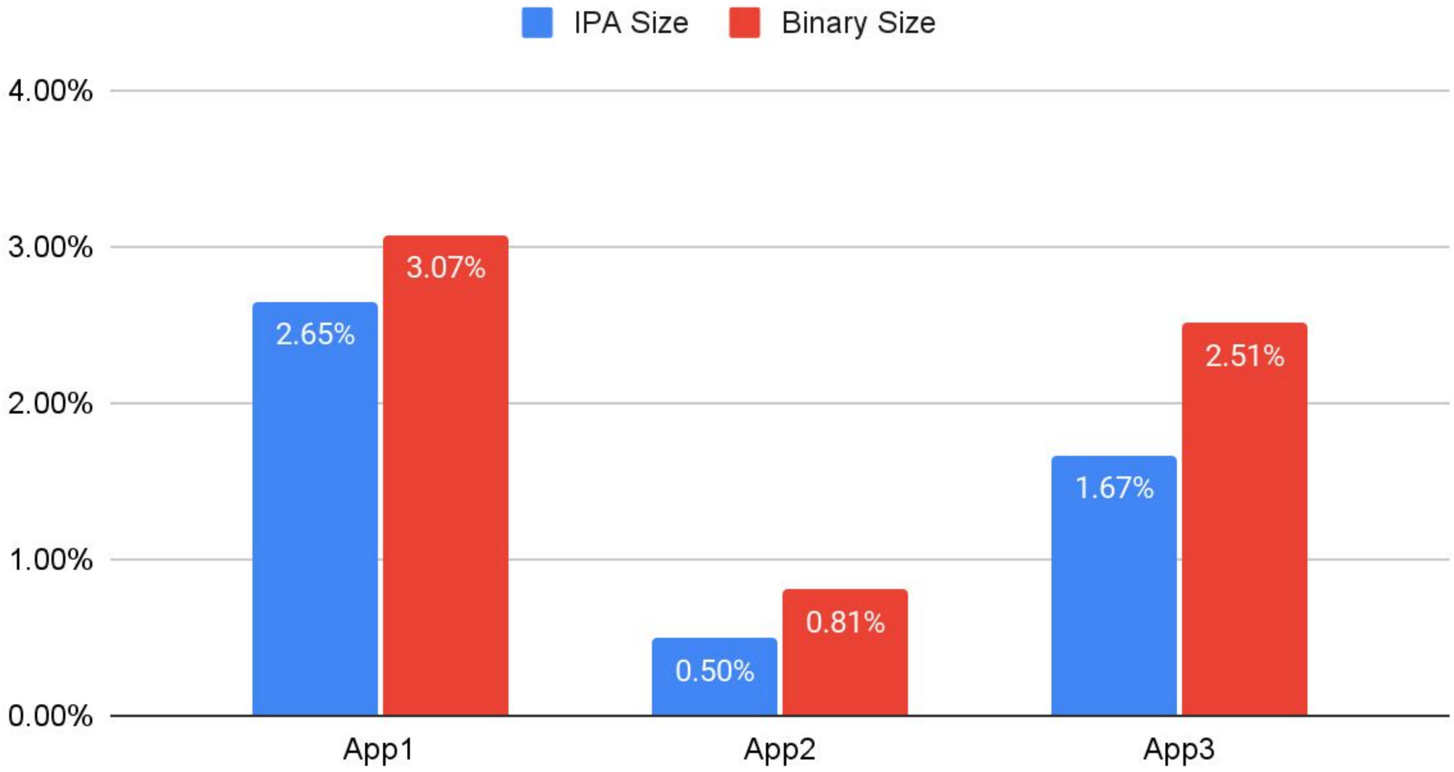
# Results

- 3 iOS apps
  - ~80% total size is binary
  - ~20% hot functions
- Firefox for iOS
  - https://github.com/mozilla-mobile/firefox-ios
  - Only controlled `Client` binary
    - 30% total size

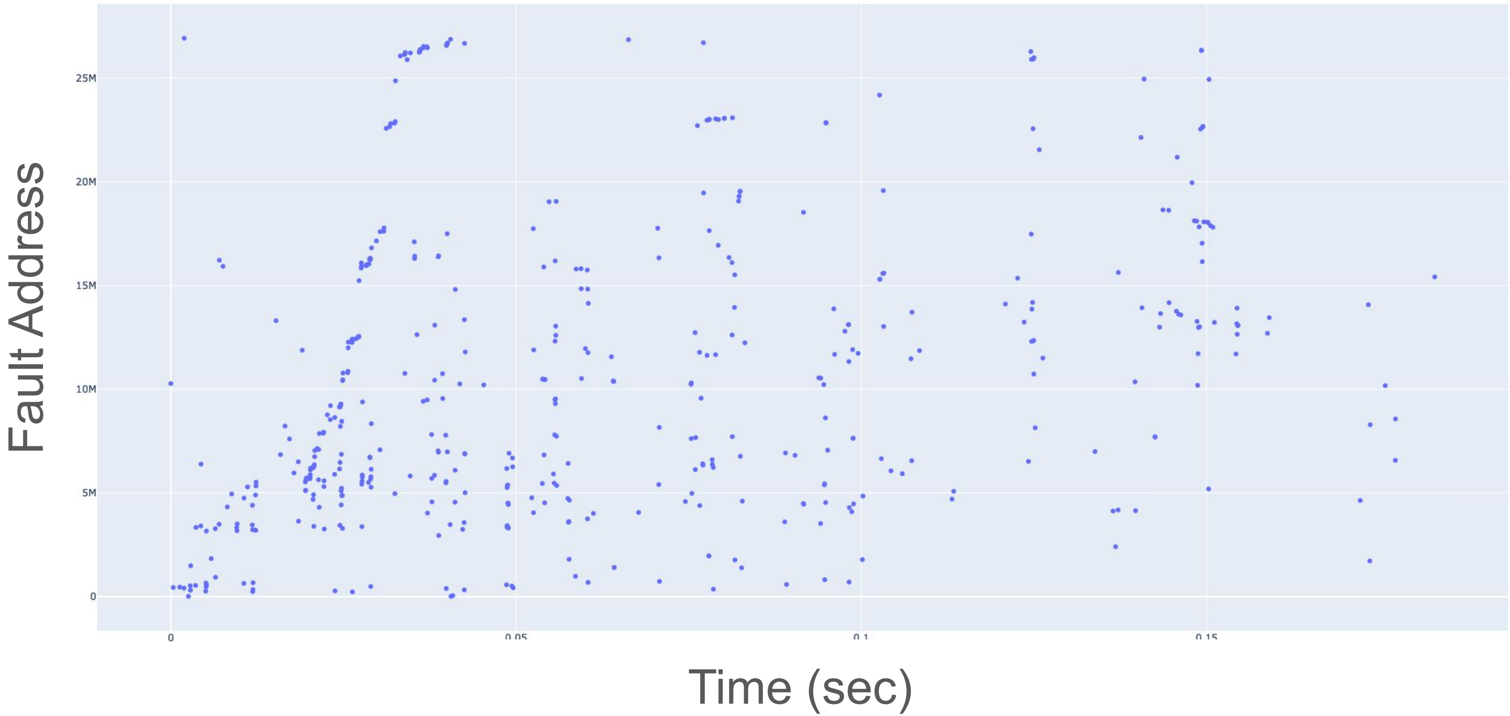
## Compressed Size Improvement (w/o Traces)



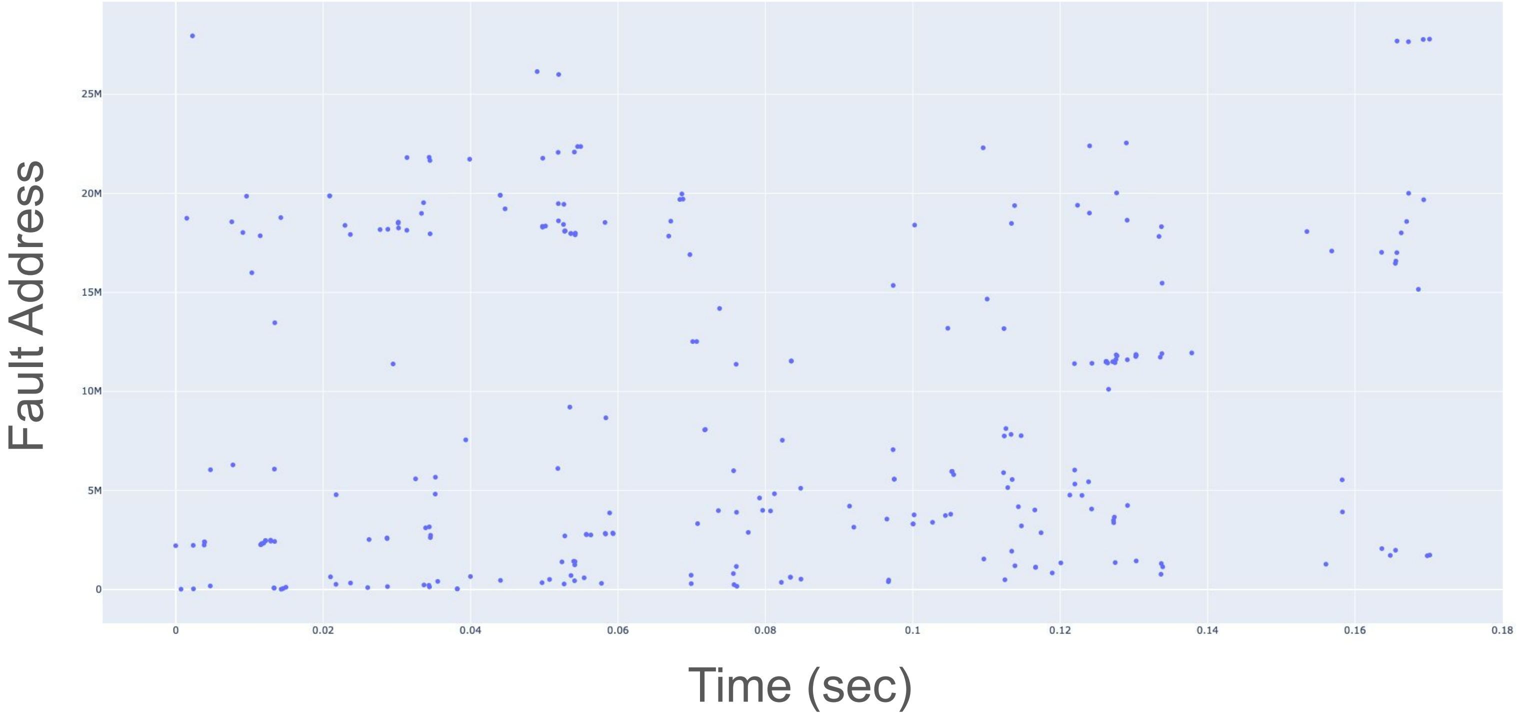
# Compressed Size Improvement (w/ Traces)



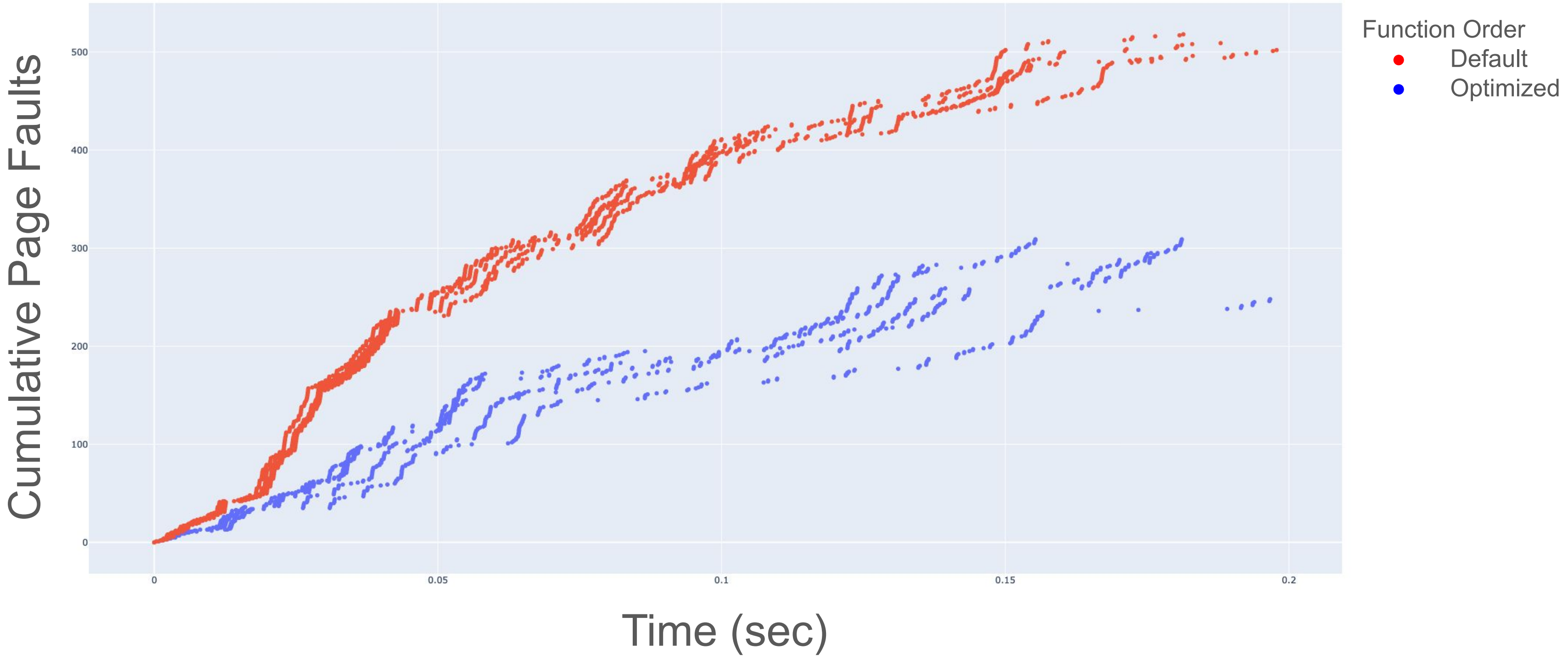
# Text Segment Page Faults (Original)



# Text Segment Page Faults (Optimized)



# Cumulative Text Segment Page Fault Count



# Conclusion

- Temporal Profiling
- Balanced Partitioning
- 40% fewer page faults
- 0.8 - 3% smaller compressed size
- Future work
  - Order **data** sections
  - Profile guided outlining
  - ???