Optimizing the Linux Kernel using AutoFDO & Propeller

Rong Xu & Han Shen

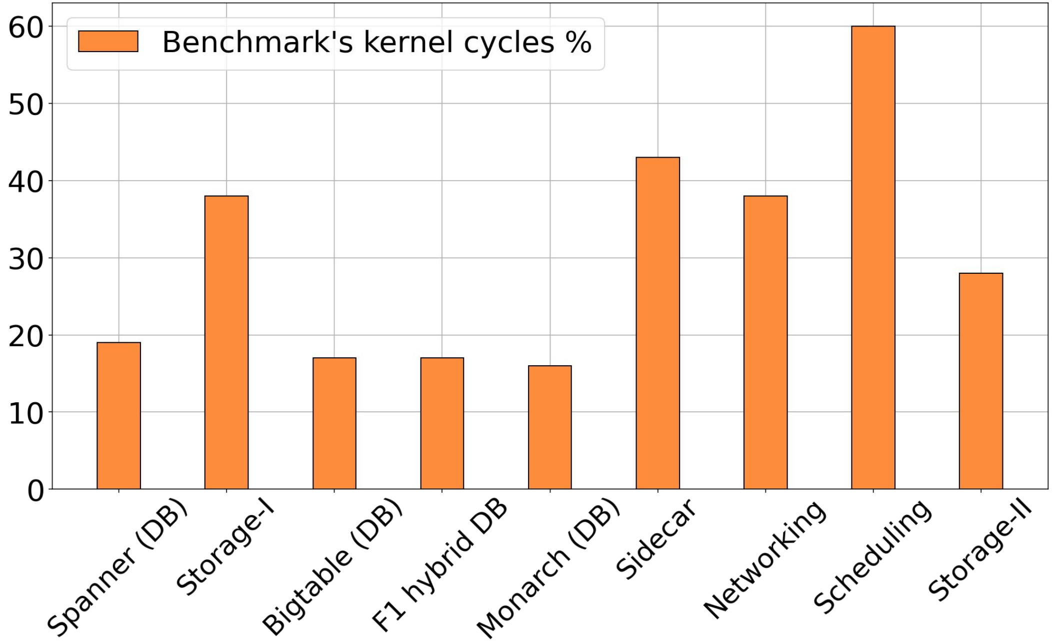
Contributors: Sriraman Tallam Krzysztof Pszeniczny Xinliang (David) Li Luigi Rizzo Nick DeSaulniers

- Build kernel with FDO (iFDO and AutoFDO)
  - Overview
  - Experimental results
- Build kernel with AutoFDO and Propeller
  - Overview
  - Experimental results

- Build kernel with FDO (iFDO and AutoFDO)
  - Overview
  - Experimental results
- Build kernel with AutoFDO and Propeller
  - Overview
  - Experimental results

Fraction of cycles spent in the kernel

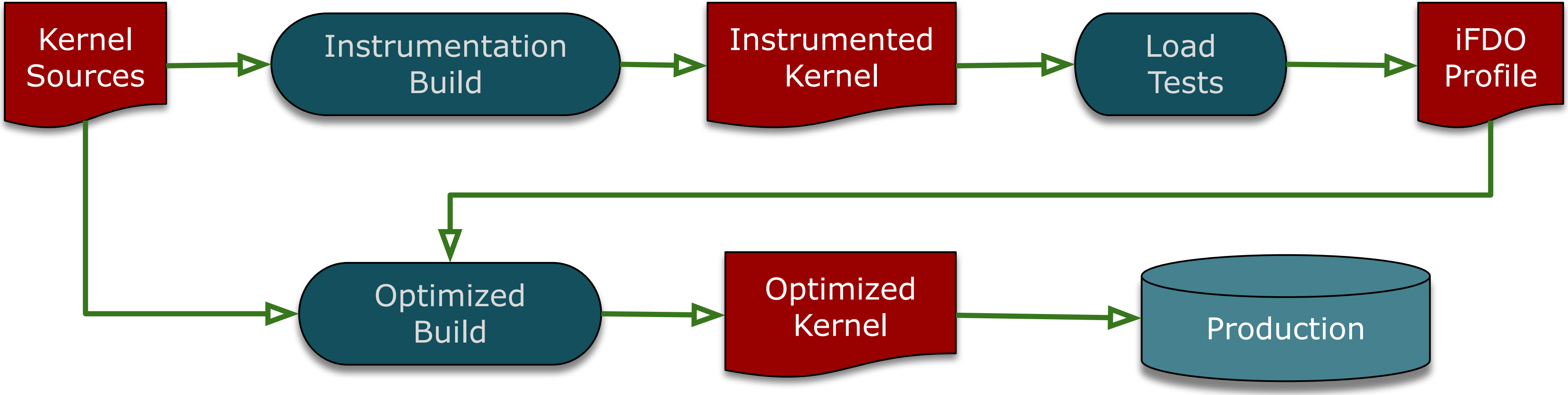
- Data center applications spend significant % of cycles in the kernel



FDO (Feedback Directed Optimization)

- Leveraging runtime insights for improved compiler codegen
- Core idea:
  - Gathers profiling data from real program executions
  - Uses this data to guide optimization decisions within the compiler
  - Focuses on optimizations that have the most impact based on actual usage
- Proved to be effective for real world applications: up to to 20% improvement
  - Better Icache, iTLB utilization
  - Better branch performance
- Instrumentation based and Sample based

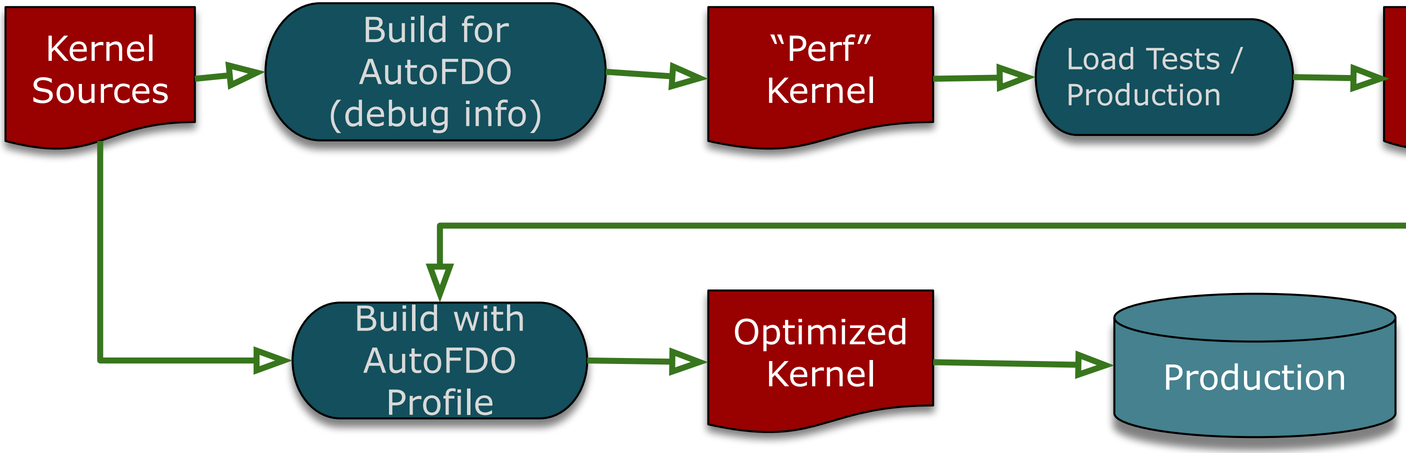
iFDO (Instrumentation based FDO)



Pros:
- Accurate profiles of the load tests
- No hardware PMU dependency

Cons:
- Instrumented binary is slow
- Needs kernel source support
- Maintain non-instrumentable file list
- Need representative load-tests

AutoFDO (Sample based FDO)

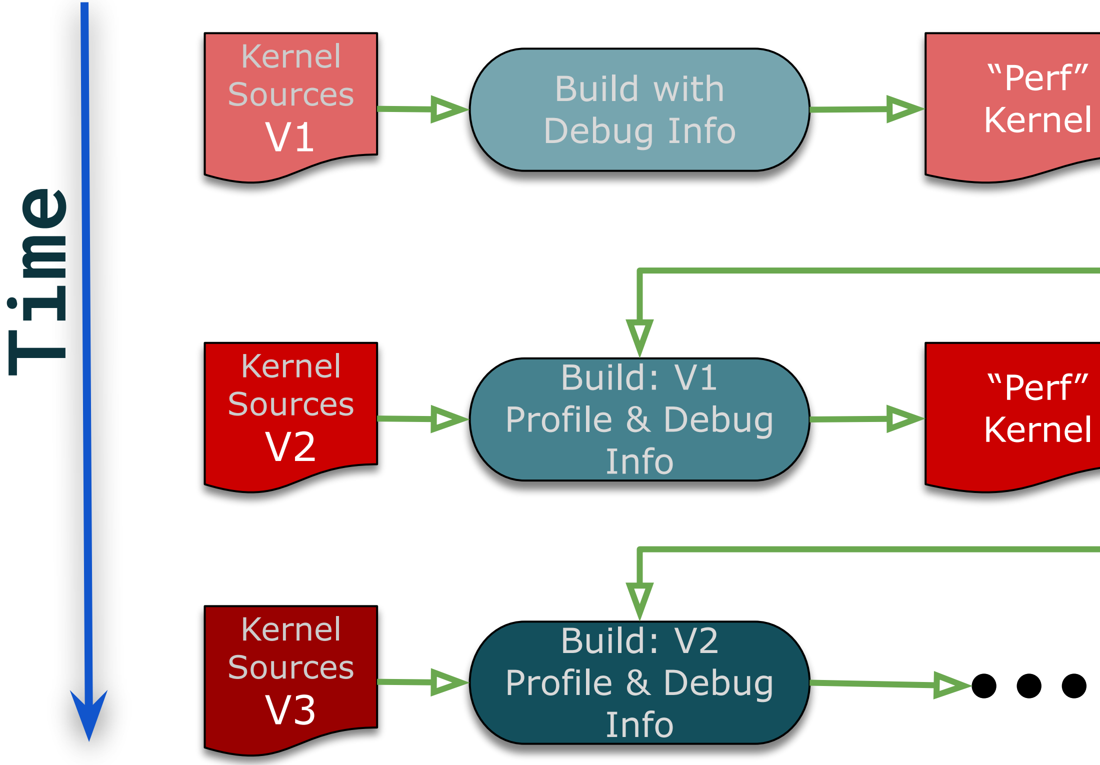


"Perf" kernel : Same codegen as non-AutoFDO build with extra debug info.

Pro:
- Very low overhead profile collection
- Production Profiles – Representative

Con:
- Lower performance using load-tests
- Requires hardware LBR and perf support

Iterative release mode for AutoFDO



LBR for AutoFDO & Propeller

- Intel LBR like hardware support
- Offline tools to convert LBR data to profile
  - create_llvm_prof or llvm_profgen

AMD
- Zen3: BRS
- Zen4: LBR-v2

ARM
- ETM
- SPE

Time

LBR Entries:

SRC_ADDR_**I** → DST_ADDR_**J**

SRC_ADDR_**A** → DST_ADDR_**B**

SRC_ADDR_**P** → DST_ADDR_**Q**

SRC_ADDR_**X** → DST_ADDR_**Y**

Increment Ranges:

[Y … P], [Q … A] , [B … I]

Increment Jumps:

X → Y, P → Q, A → B, I → J

AutoFDO Profile

```
Func:total_samples:head_samples
  Offset1: num_samples
  Offset2: num_samples

foo:42:9
 0: 8
 2: 10 bar:4
 7.2: 6
 30: goo:8
  1: 4
  5: boo:4
   2.1: 1
   2.2: 3
 41: 10
```

List of optimizations that benefit from FDO

- Function inlining:
  - Removing call overhead
  - Enlarging optimization scope that matters
- BasicBlock layout: increase branch fall-through
  - Fall-through is just more effective than taken even both correctly predicted
  - Fall-through groups more hot BBs together -- better i-cache utilization
- Indirect-call promotion (value profiling)
  - Reducing indirect-call and making inline possible
- Other optimizations:
  - Function layout
  - Machine function splitting
  - Scalar optimization, like speculative PRE
  - Loop nested optimization, like unrolling / peeling and vectorization
  - Partial inlining
  - Register allocation
  - ThinLTO

- Build kernel with FDO (iFDO and AutoFDO)
  - Overview
  - Experimental results
- Build kernel with AutoFDO and Propeller
  - Overview
  - Experimental results

Experiment results

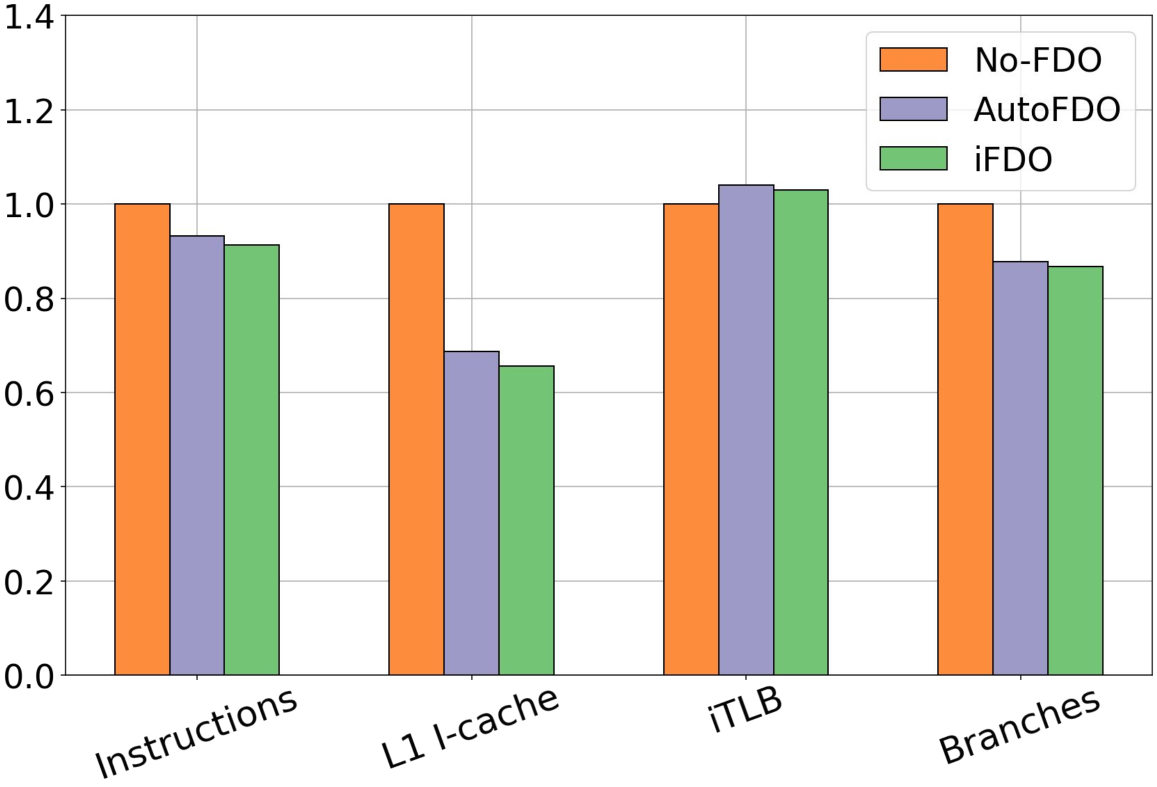
Micro-Benchmarks:

Benchmark	Metrics	AutoFDO improvement	iFDO improvement
Neper / tcp_rr	Latency improvement (geomean of P1/P50/P99/Mean)	10.6%	11.8%
Neper / tcp_stream	Throughputs improvement	6.1%	6.7%
UnixBench (1-instance)	Index score	2.2%	3.0%
UnixBench(112-instance)	Index score	2.6%	2.6%

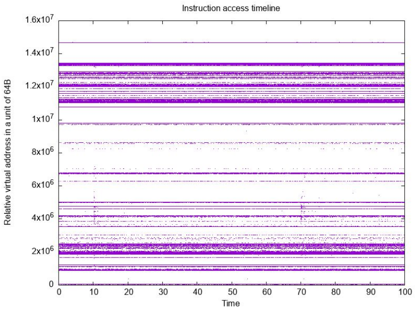
Load-tests:

- Google database app: improves 2.6% with AutoFDO kernel, 2.9% with iFDO kernel
- Meta services: improves ~5% with AutoFDO kernel, ~6% with iFDO kernel

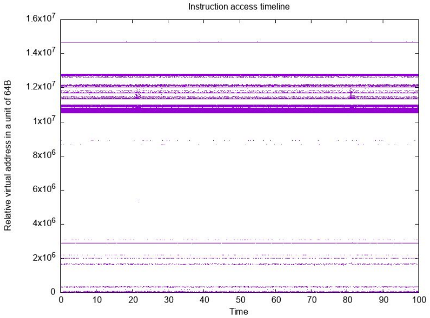
Kernel PMU stats for tcp_rr



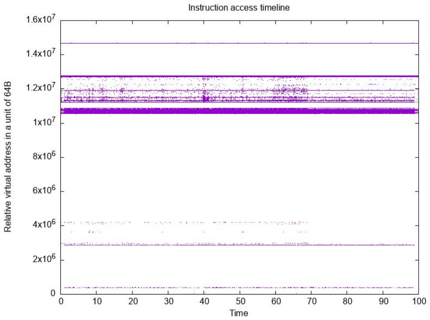
Instruction heatmap comparison (tcp_rr)



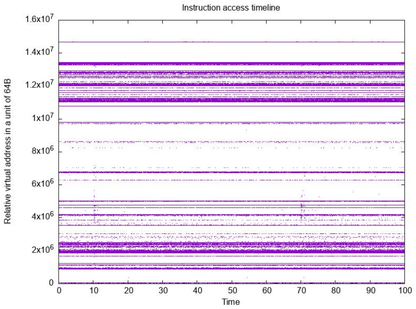
nofdo



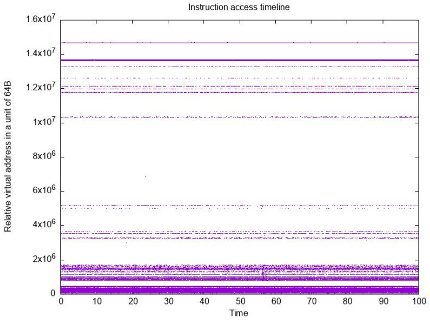
ifdo



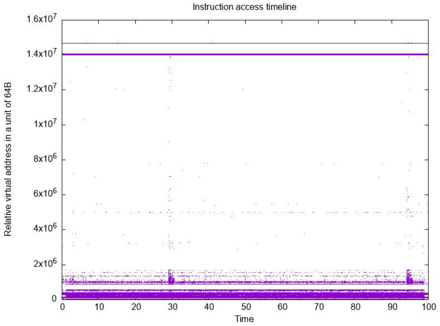
ifdo + thinlto



nofdo



autofdo



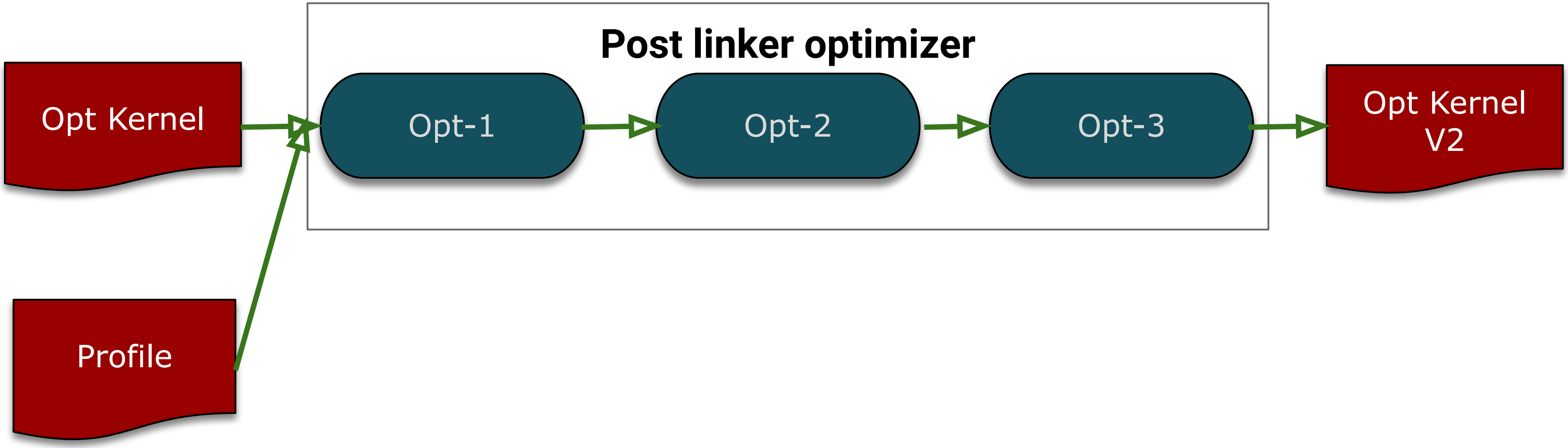
autofdo + thinlto

AutoFDO kernel: lessons, challenges and TODOs

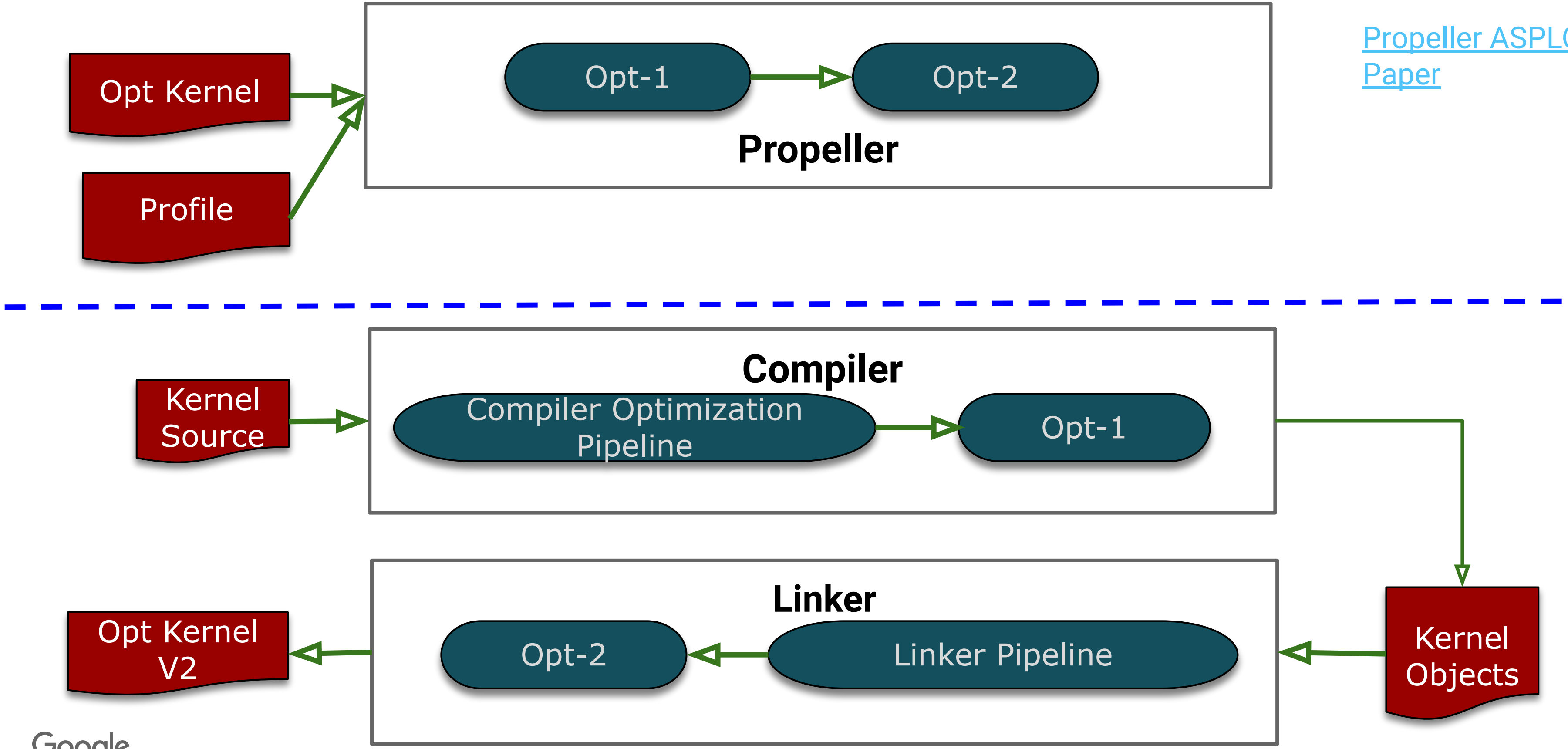
- Improve offline tools to support kernel
- Things to be done:
  - Apply unique linkage names for static functions
  - Module support
- Testing is most challenging
  - Fast and reliable performance test
  - Representative workloads
- Lessons:
  - AutoFDO is easy to use
  - System with sufficient load during profiling
  - Intel machine profile works well on AMD machines
  - Customized kernel helps the performance
  - Enable LTO / ThinLTO for better performance

- Build kernel with FDO (iFDO and AutoFDO)
  - Overview
  - Experimental results
- Build kernel with AutoFDO and Propeller
  - Overview
  - Experimental results

What is a post linker optimizer?



Propeller in practice

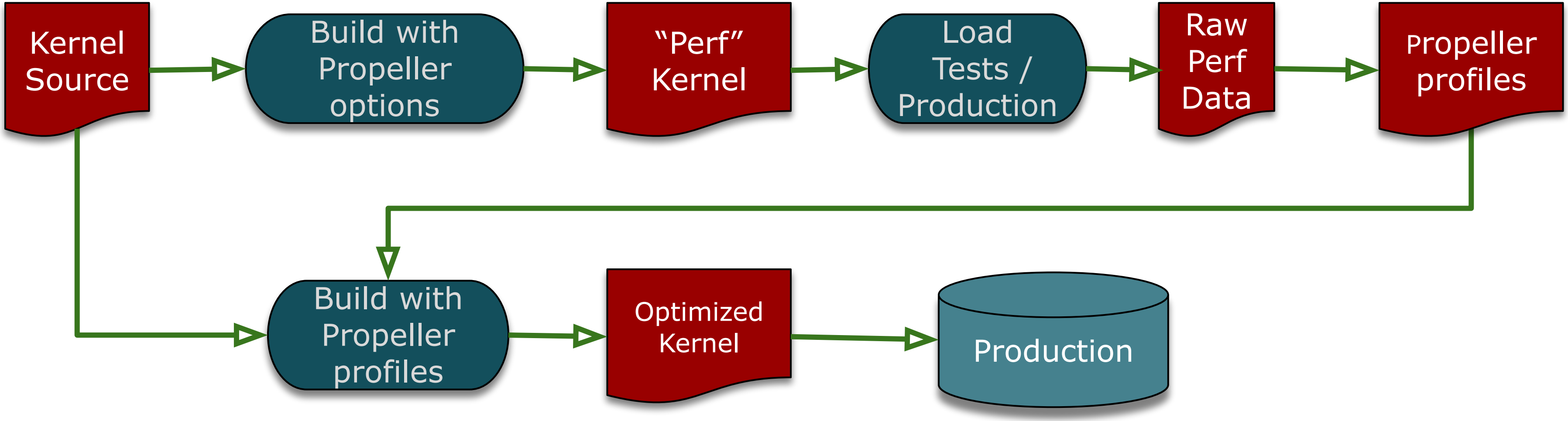


Propeller ASPLOS Paper

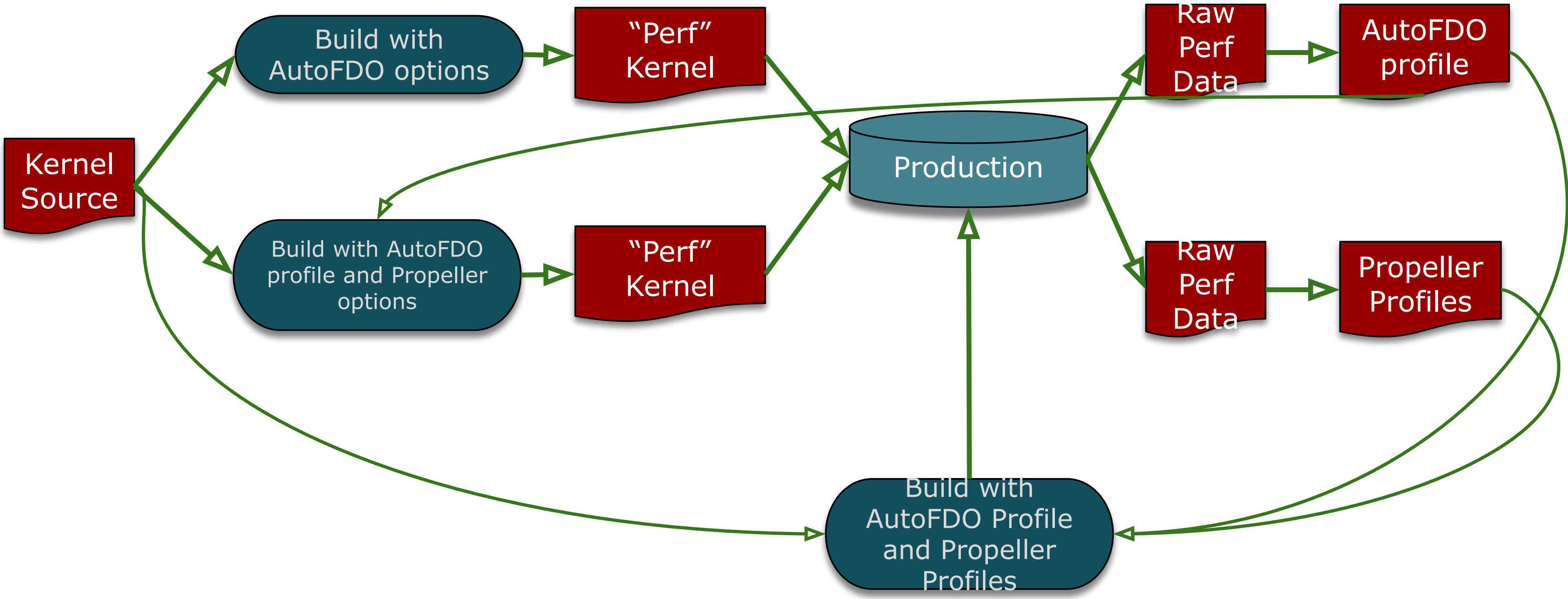
Propeller - Cont'd

1. Pros
   - ❑ Avoid disassembly
   - ❑ Scalable for distributed build systems, warehouse-scaled application ready
2. Cons
   - ❑ Requires sources and needs to re-build the binary
3. Optimizations
   - ❑ Basic block layout (details)
   - ❑ Path cloning (details)
   - ❑ Inter-procedure register allocation - work in progress (details)

Propeller in Action



Propeller on AutoFDO



Additional Notes on Propeller

- Propeller and kernel modules
  - Currently, post linker optimizers work for executables and shared libraries.
  - Kernel modules are not executables nor shared libraries, they are relocatable objects.
  - Propeller is capable of optimizing kernel modules too.
- Propeller and unique static function names in kernels
  - “-funique-internal-linkage-names” cannot be used
- Propeller works well with iFDO, ThinLTO.

Additional Notes on Propeller - Cont'd

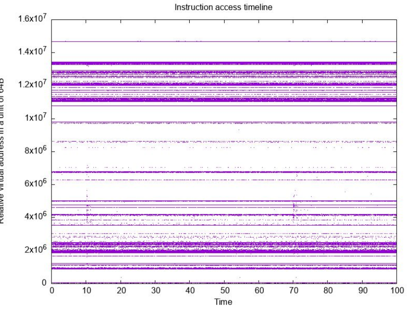
- AutoFDO profiles offer flexibility by tolerating:
  - Source code variations: Minor changes in the source code won't necessarily invalidate the profile.
  - Different build options: Slight adjustments to build settings can still be compatible with the profile.
- Propeller profiles has zero tolerance for source code or build settings changes
  - The source code and build settings must be identical to those used during profile generation.

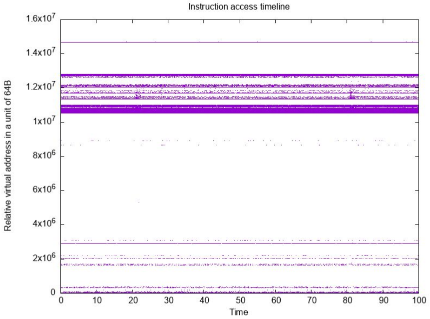
Propeller - toolings and supported platforms

1. Requires github.com/google/autofdo
2. Requires Hardware support
   a. On X86_64
      i. LBR - INTEL Haswell (16-Entry LBR) or INTEL Skylake (32-Entry LBR) or later
      ii. BRS - AMD Zen 3 EPYC
      iii. LBR EXT V2 - AMD Zen 4
   b. On Arm
      i. Arm SPE
      ii. Arm ETM

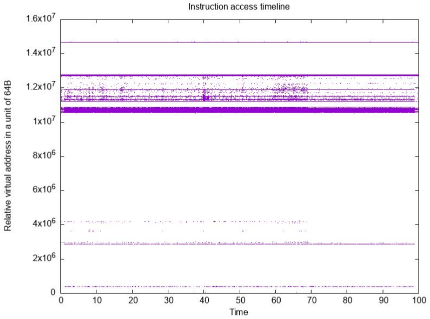
Fully validated?	Kernel	Internal Applications
LBR	✓	✓
BRS		✓
LBR EXTV2		
Arm SPE		✓
Arm ETM		

Instruction heatmap comparison (tcp_rr)

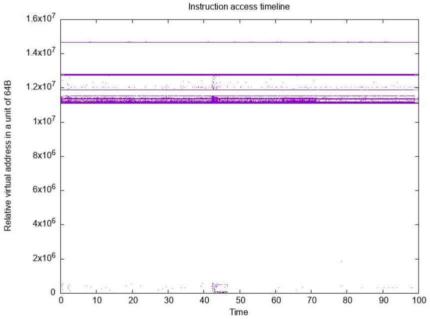




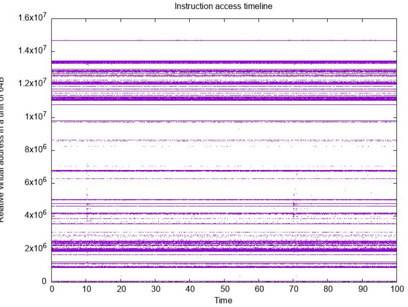
ifdo

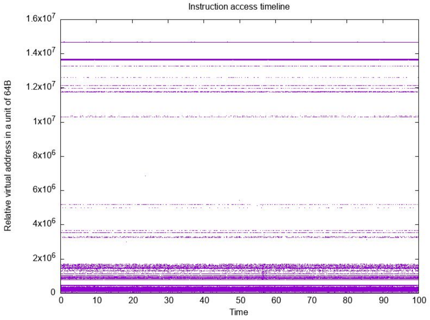


ifdo + thinlto

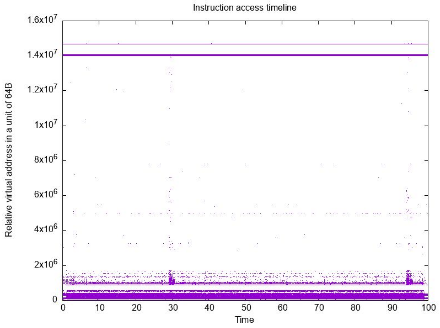


ifdo + thinlto + propeller

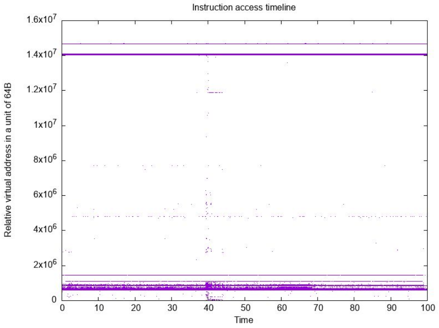




autofdo



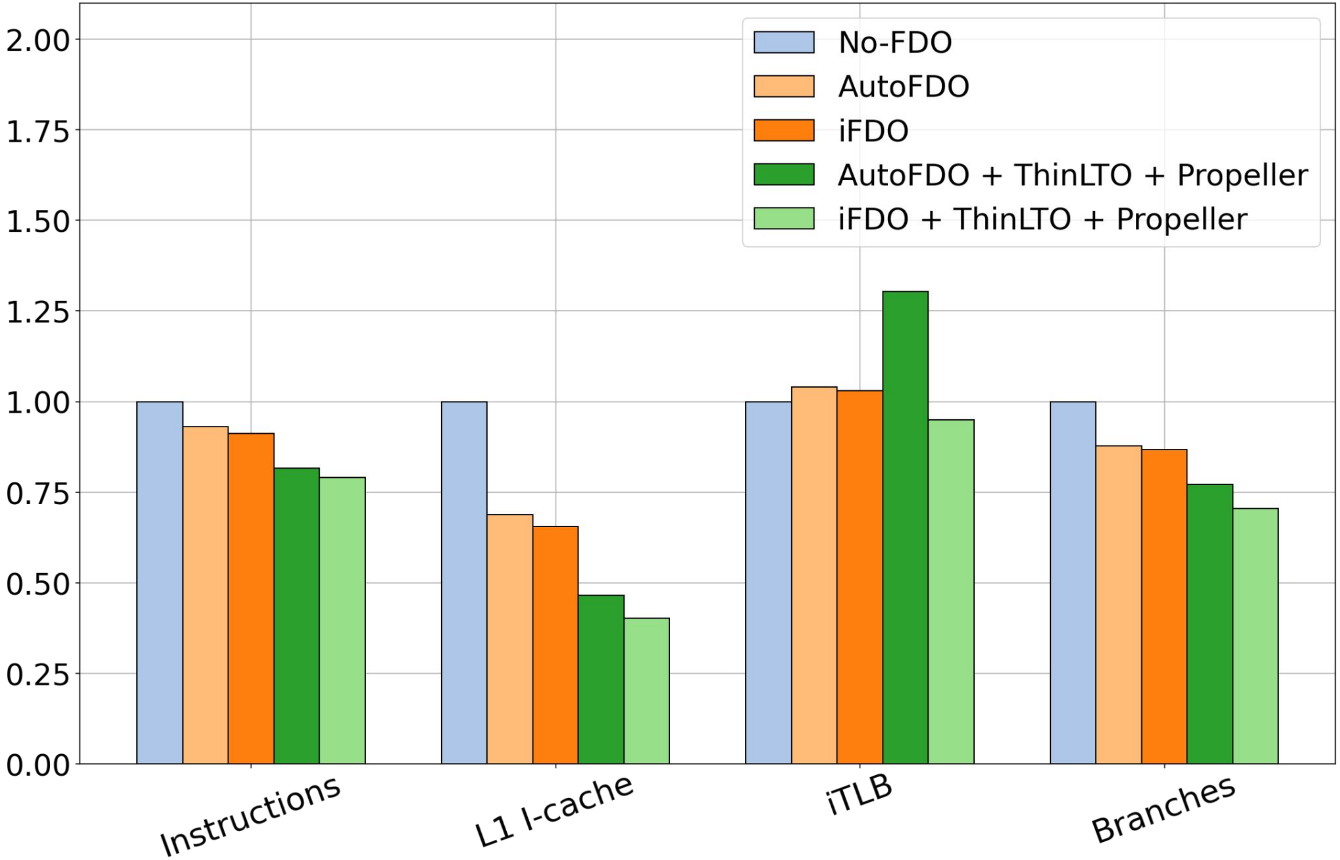
autofdo + thinlto



autofdo + thinlto +propeller

Google

Kernel PMU stats for tcp_rr



Current status

- Patches submitted for review
- Internally doing large scale production tests to measure the performance
- Investigating customized kernel based on specific workload

Summary

- FDO improves kernel performance significantly
- AutoFDO can integrate with kernel build very well
    - Easier to deploy
    - Can be profilied from production
    - Get most of the iFDO performance or even better
- Add Propeller to get best possible performance

Thank you!