Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications

Han Shen*
Google
USA
shenhan@google.com

Krzysztof Pszeniczny*
Google
Switzerland
kpszeniczny@google.com

Rahman Lavaee*
Google
USA
rahmanl@google.com

Snehasish Kumar*
Google
USA
sneaky@google.com

Sriraman Tallam*
Google
USA
tmsriram@google.com

Xinliang David Li*
Google
USA
davidxl@google.com

- **Post Link Optimizers: Why and What?**
- Why doesn't state-of-the-art measure up?
- How have we made Propeller scale?

中山大學
SUN YAT-SEN UNIVERSITY

PGO is great but there is room!

- Peak Optimized binaries built with PGO and ThinLTO
  - Profile Guided Optimizations (PGO): > 10% on production binaries
  - Cross-module optimizations with ThinLTO:-3% production binaries
- PGO passes suffer from profile imprecision
  - 3% performance dropped, imprecise code layout!
- Binary level optimizations cannot be done
  - DSB alignment misses
  - Inter-procedural Register spill removal, caller-callee semantics
  - Inserting Prefetch Instructions

Post Link Optimizers

- What is a Post Link Optimizer? (Compared to Regular PGO)
  - Does not re-compile, directly optimize the binary
  - Disassemble, optimize and rewrite, eg: META's BOLT
- Targeted Optimizations
  - Code Layout recovers the lost performance due to profile imprecision
- Profile Guided: Precise profiles on a highly optimized binary
  - Context and Flow Sensitive
- New class of optimizations are possible
  - Precise Prefetch Insertion, Code Alignment,...

- Post Link Optimizers: Why and What?
- **Why doesn't state-of-the-art measure up?**
- How have we made Propeller scale?

中山大學
SUN YAT-SEN UNIVERSITY

Build Scalability - Peak RAM hard limits

- Google's Distributed Build System
  - 5M binary and test builds per day, 15M actions executed daily
  - 50K+ developers and 3 billion LOC, fast turnaround times
  - Artifacts: Content based caching, 90% hit rate
  - 12G memory limit for most processes
- Chromium Builds
  - 10G limit per linker process for ThinLTO
- Debug Fission
  - Goals: Scalability, Incremental Builds
- ThinLTO
  - Scalable LTO
  - Whole Program Optimization: Enabled broader adoption

State-of-the-art does not scale!

- Single Process, Multi-threaded Optimizers don't scale with binary sizes!
- Do not take advantage of Distributed Build Systems
  - Extremely painful to re-engineer & distribute these actions

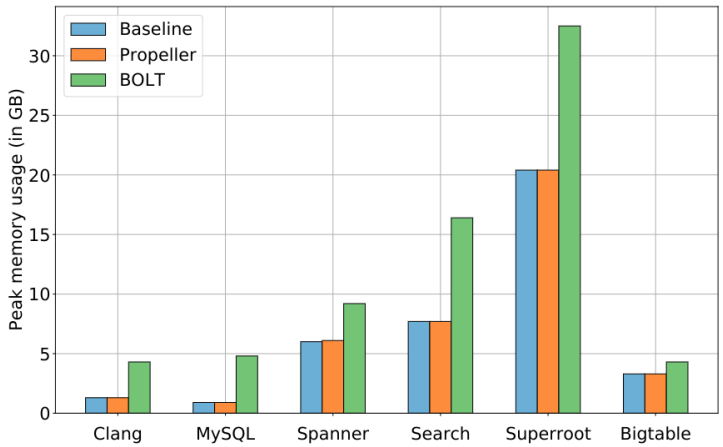

Figure: Peak memory usage for Propeller (Phase 4), BOLT optimizations and baseline link action.

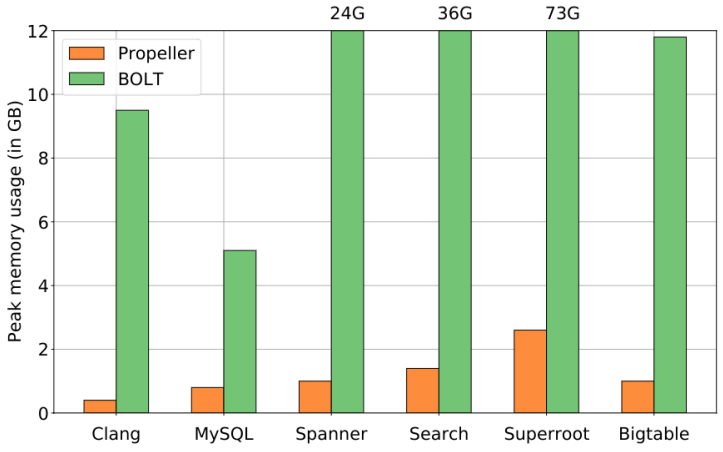

Figure: Peak memory usage during profile conversion and whole program analysis.

Disassembly and Rewrite Issues!

- Standalone Binary Optimizers re-engineer the tool chain
  - Redundancy: Debug Info Parse & Rewrite, CFI Parse & Rewrite
  - Downstream & not integrated, ABI & tool-chain compatibility
- BOLT doesn't honor RSEQ (Kernel Restartable Sequences)
- Cryptographic modules should not be rewritten
  - Strict startup integrity checks
- BOLT binaries couldn't be stripped
  - Misaligned PT_LOADS crashes binary
  - Upstream Arguments on whether this is a ABI problem

- Post Link Optimizers: Why and What?
- Why doesn't state-of-the-art measure up?
- **How have we made Propeller scale?**

Propeller Design Philosophy

- Guiding Principles
  - No Disassembly, Distributed Actions => Memory Overheads below thresholds
  - Integrated with the tool chain, not a standalone tool!
  - Lightweight Whole Program Analysis is the key!
- RELINK as opposed to REWRITE
  - Summary based Analysis-Inspired by ThinLTO
  - Build Artifacts cached, no disassembly required
  - Relink previous optimized IR after applying post link transformations (Backend Actions)
- Backend Actions can be distributed
  - Naturally fits with distributed build systems
- Framework for Post Link Optimizations
  - Currently, only code layout is available
  - Working on Inter-proc. Register allocation, code prefetching

Propeller Design

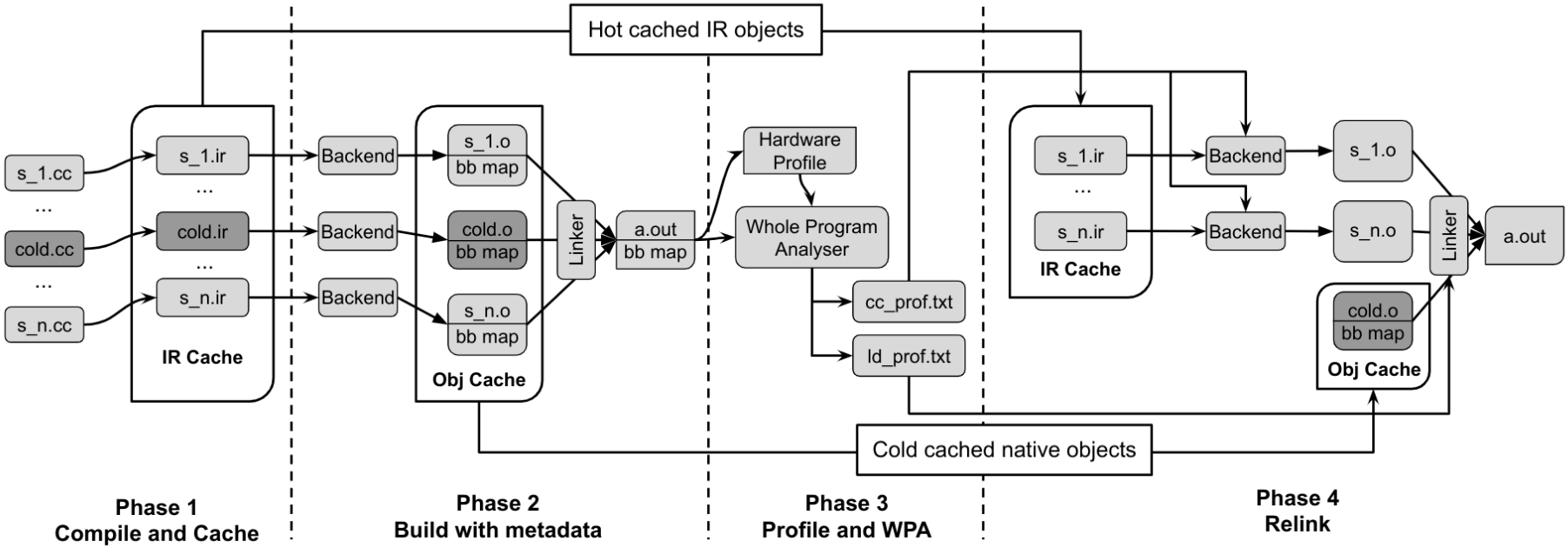

Figure 1: Design of a Profile Guided, Relinking Optimizer

Basic Block Sections - Easy Code Layout

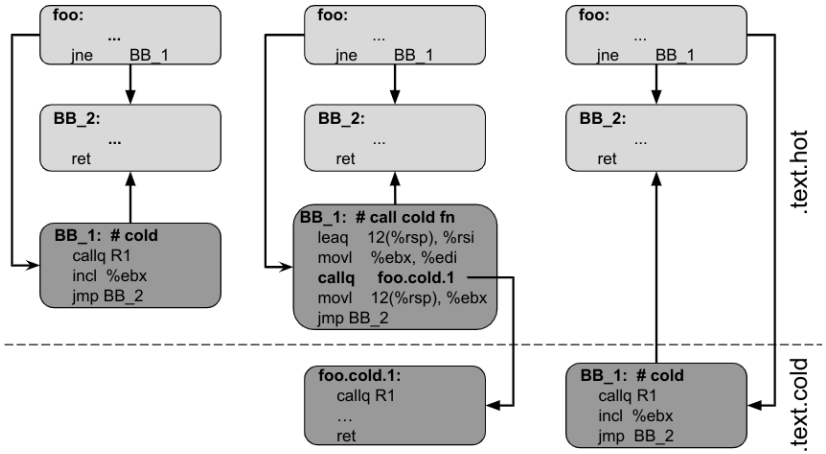

Figure 2: Original function layout (L), Splitting via function call (C), Splitting with basic block sections (R)

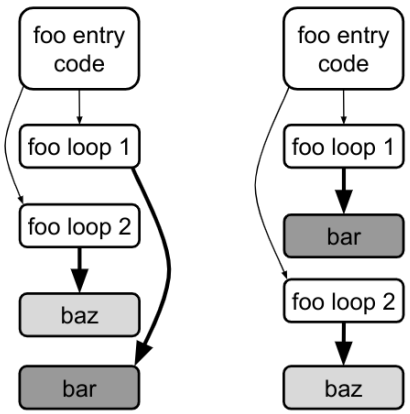

Figure 3: Optimal intra-function (L) and inter-function (R) layout. Hotter edges are shown by thicker lines.

Performance Improvements

Table 3: Performance improvements of Propeller and BOLT optimized binaries over PGO and ThinLTO.

Benchmark	Metric	% Improvements	
		Propeller	BOLT
Clang	Walltime	7.3 %	7.3 %
MySQL	Latency	1 %	0.8 %
Spanner	Latency	7 %	*Crash*
Search	QPS	3 %	4 %
Superroot	QPS	1.1 %	*Crash*
Bigtable	QPS	3 %	*Crash*

Peak RAM Build Actions

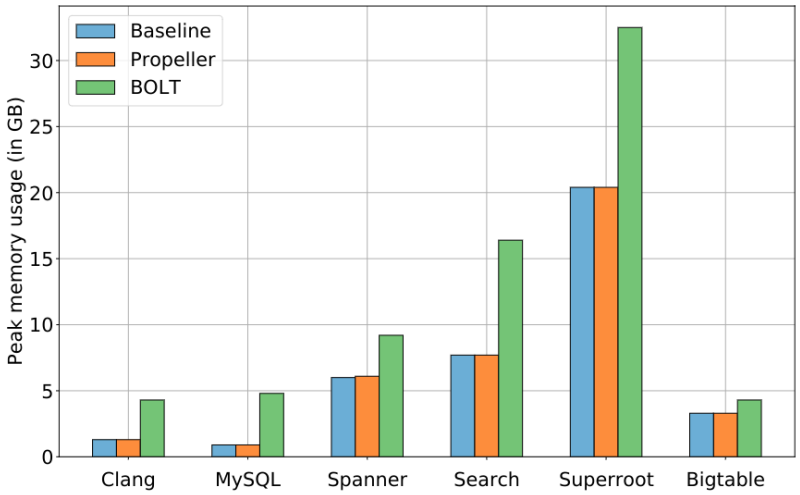

Figure: Peak memory usage for Propeller (Phase 4), BOLT optimizations and baseline link action.

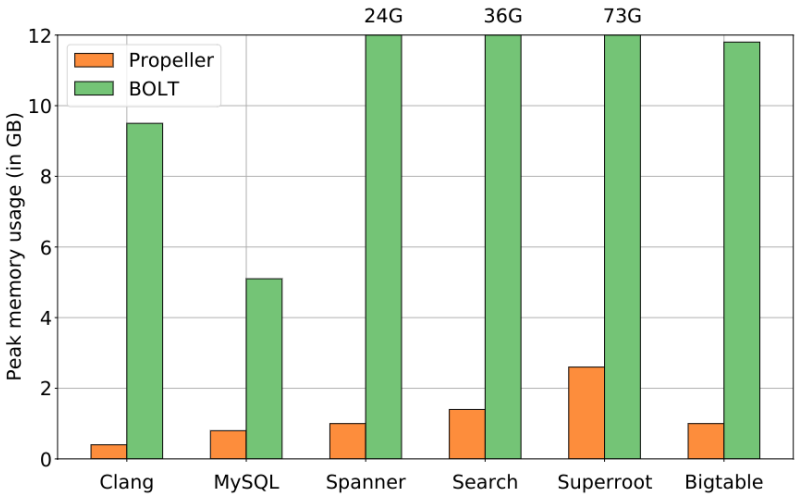

Figure: Peak memory usage during profile conversion and whole program analysis.

Final Thoughts - Hardware Software Codesign

- Post Link Optimizers can play an important role
- How can the PMU be enhanced to provide more useful information?
  - Last Branch Record (LBR) has been a game changer
  - What other counter information can we use?
  - Lightweight Whole Program Analysis is the key!
- Prefetch and Cache line demote instructions
- DSB: more information to guide precise alignment
- Branch Prediction
  - Hardware provides more information to guide optimizations
  - Compiler inserts hints that in branch instructions that predictors can exploit