Smaller and Faster Android

— A talk from a practitioner to fellow ones

Shih-wei Liao
廖世偉
sliao@google.com

Google™

Systematic Optimizations

- We’re asked to present:
  - data-driven,
  - systematic

  studies in optimizing:
  - size,
  - performance,
  - power

  to Taiwan's Android community (Brand+ODM+OEM)

Google™

Outline: 7 Optimization Strategies

- **Data-driven tool deployment:**
  - Regularly evaluate & then leverage the winner among optimizing toolchains
- **Judicious abstraction & specifications**: A fundamental methodology
  - *Visibility* of a function should match the API spec in programmer's design
  - Tradeoff in splitting into Java and Native: This interface affects performance
  - Opencore → OpenMax: Taiwan's IC industry looks for APIs to differentiate
- **Systematic parameter setting**: A key driver in performance/size
- **Profile-guided optimizations**: A useful methodology
  - Feedback-Directed Optimizations (FDO): Build-Run-Build with our arm-eabi-gcc
  - Class loading profiler (aka Preload profiler): Zygote's preloading: Trade-off between boot-up time and app init time.
- **Scope-enhancing optimizations**: Interprocedural optimizations via arm-eabi-gcc --fripa
- **Redundancy elimination**: Identical Comdat (or Code) Folding (ICF)
- **Memory management** optimization in Dalvik

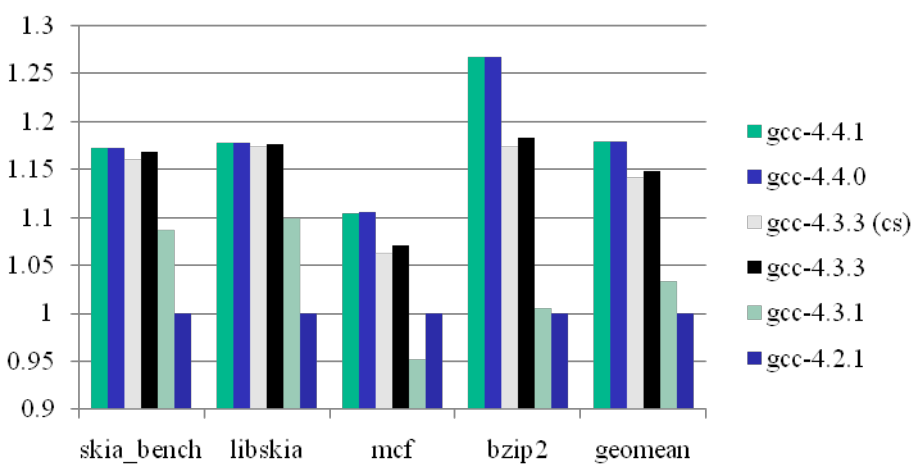
In the interest of time, the talk only covers 1st item each of the 7 ways

1. Data-driven tool deployment

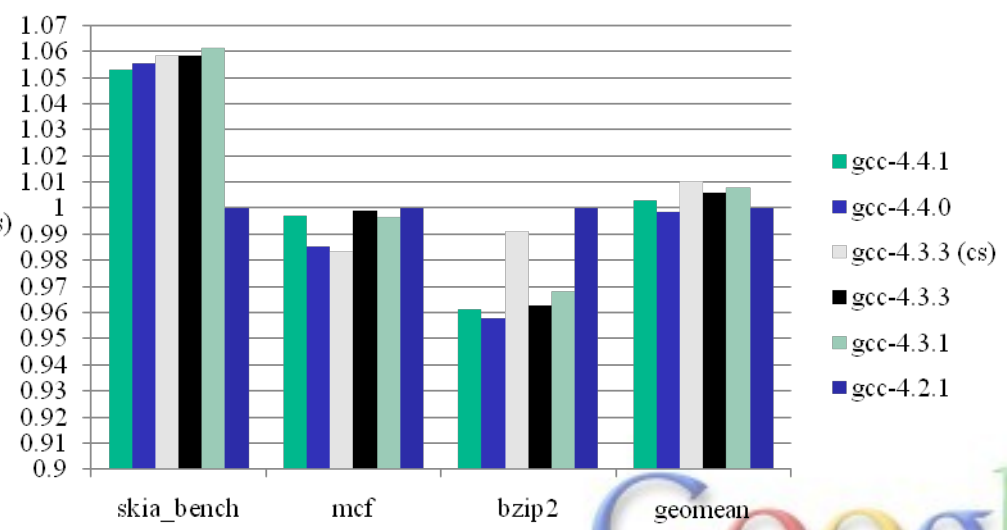
- Analyze the tools candidates:

gcc-4.4.1	gcc-4.4.0	gcc-4.3.3 (cs)	gcc-4.3.3	gcc-4.3.1	gcc-4.2.1
Without Code Sourcery 2009q3	Android's toolchain for Éclair	Code Sourcery 2009q1	Without Code Sourcery 2009q1	Android's nondefault toolchain for Donut	Android's toolchain for Donut

Size improvement on Dream phone



Speedup on Dream phone (Run 100x)



We track 13 numbers daily.
We got space to show 4 here.

Google™

Analyze 6 Toolchains

- Based on Android perflab benchmark results,
  - Baseline: Current Android's default toolchain: gcc-4.2.1
  - Size:
    - Both gcc-4.4.1 & gcc-4.4.0: 17.5% improvement
    - Both gcc-4.3.3 & gcc-4.3.3 Code Sourcery Version: 15% better
    - gcc-4.3.1: 3% improvement
  - Performance: No significant variance among 6 toolchains
    - gcc-4.4's size benefit comes with no performance penalty
- Code Sourcery for ARM doesn't have significant performance/size benefit over Android's version of gcc
  - Code Sourcery's strength: Addressing ARM's hardware errata early. We then port the fixes to gcc-4.4.0
- gcc-4.4 wins → Toolchain should move to 4.4 soon; Skipping 4.3 → Next slide

Google™

Android Toolchain Roadmap

	Cupcake	Donut	Éclair (armv7a)
gcc	4.2.1	4.2.1	4.4.0
binutils	2.17	2.17	2.19
gdb	6.6	6.6	6.6
gmp	4.2.2	4.2.2	4.2.4
mpfr	2.3.0	2.3.0	2.4.1

- All pieces from open source
  - GNU GCC, binutils, gdb, gmp, mpfr
  - Patch for bug fixing and optimization
    - Take patches from upstream
    - Submit our patches to upstream
- Also, native developers can use Android NDK
  - http://developer.android.com/sdk/ndk/1.5_r1/index.html

Google™

Building Android Toolchain (1/2)

- Android uses Bionic C library
  - BSD license: Keeps GPL out of user's sphere.
  - Small (~200KB), fast
  - Bionic has built-in support for important Android-specific services, e.g., system properties, logging
  - Very limited support for POSIX, C++, etc
- If need libstdc++-v3:
  - Enable libstdc++-v3 when configure the toolchain. Statically link in the necessary components.

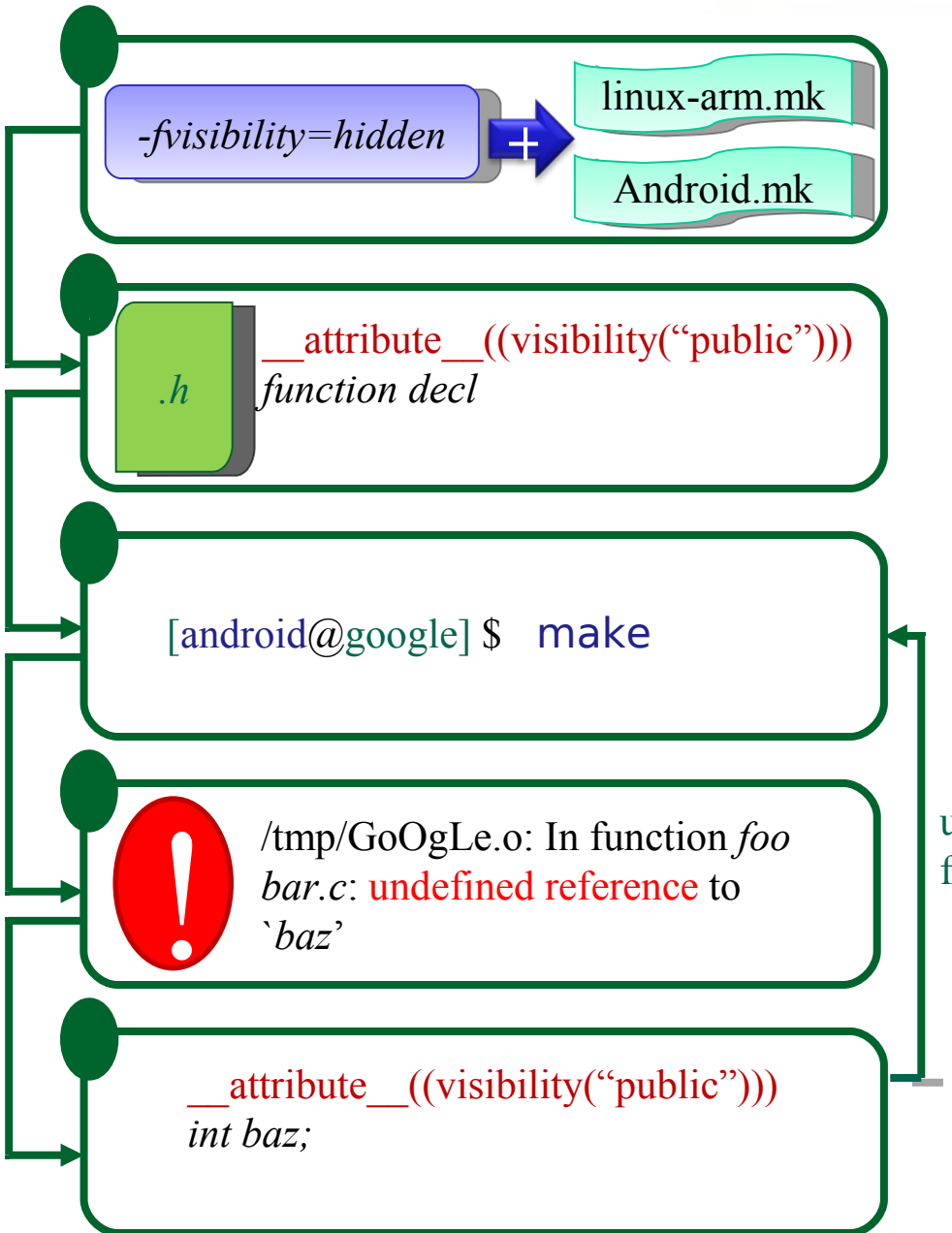
Building Android Toolchain (2/2)

- Barebone-style building:
  - Inside Android tree
  - Specify all system and bionic header file paths, shared library paths, libgcc.a, crtbegin_*.o, crtend_*.o, etc.
- Standalone-style building:
  - Our prebuilt gcc-4.4.0 toolchain
  - Convenient for native developers

    arm-eabi-gcc -mandroid --sysroot=<path to sysroot> hello.c -o hello

    (* <path to sysroot> is a pre-compiled copy of Bionic)

2. Thoughtful abstraction & specifications



Goal: *Visibility* of a function should match the API spec in programmer's design.

Solution:

First, systematically applying the 5 steps. Fundamentally, need to go through the APIs of each library:

- Consciously decide what should be "public" and what shouldn't.

Result: ~500 KB savings for Opencore libs

Key: The whole hidden functions can be garbage collected if unused locally:

- Toolchain's options: -ffunction-sections, -Wl,--gc-sections,

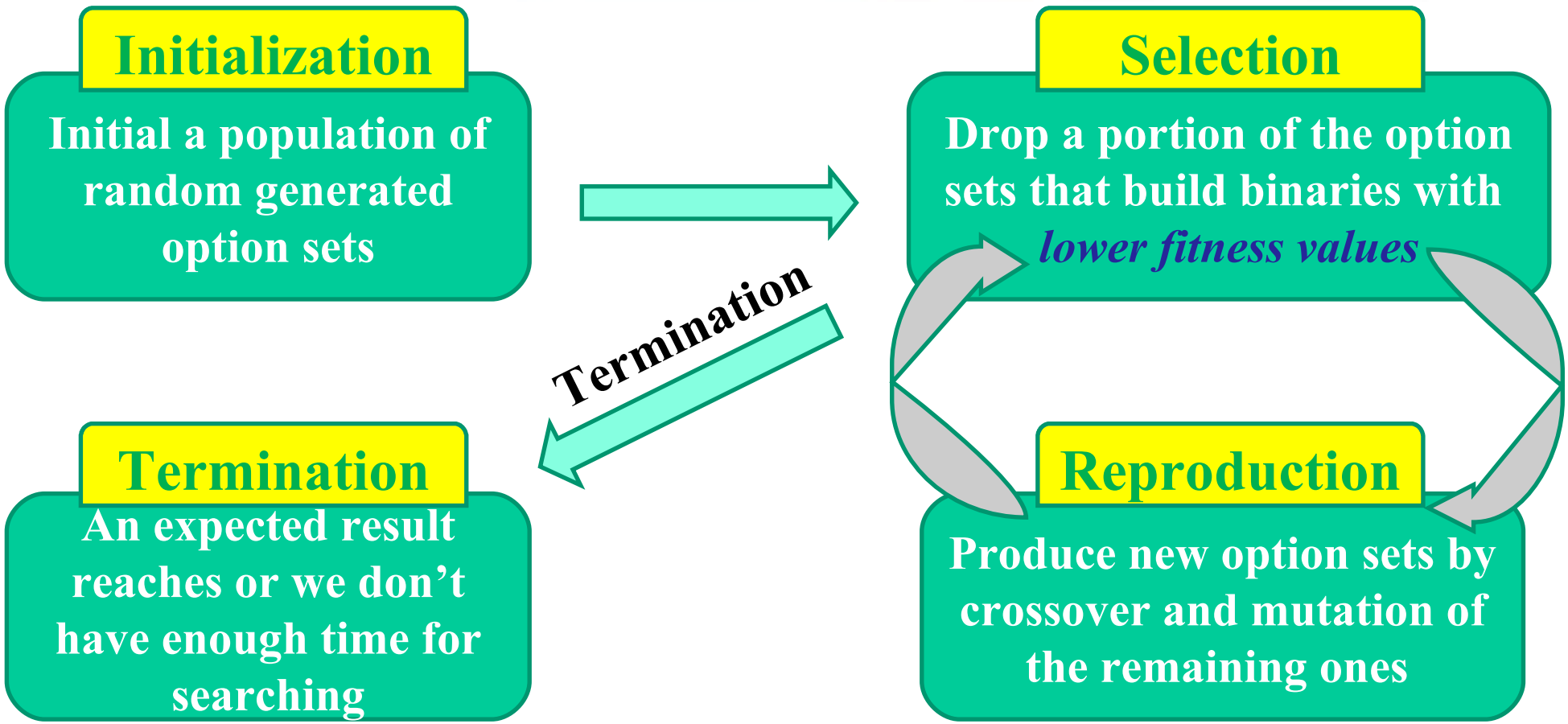
Google™

3. Parameter Setting

- Parameters setting is a key driver in performance/size optimizations
- Case study: For Android tree, find the best:
  - Compiler parameters
  - Compiler options
- Parameter space exploration via genetic algorithm (GA)

Google™

GA Search For Compiler Options



Optimization target	Fitness function
Performance	Inverse of execution time
Size	Inverse of code size

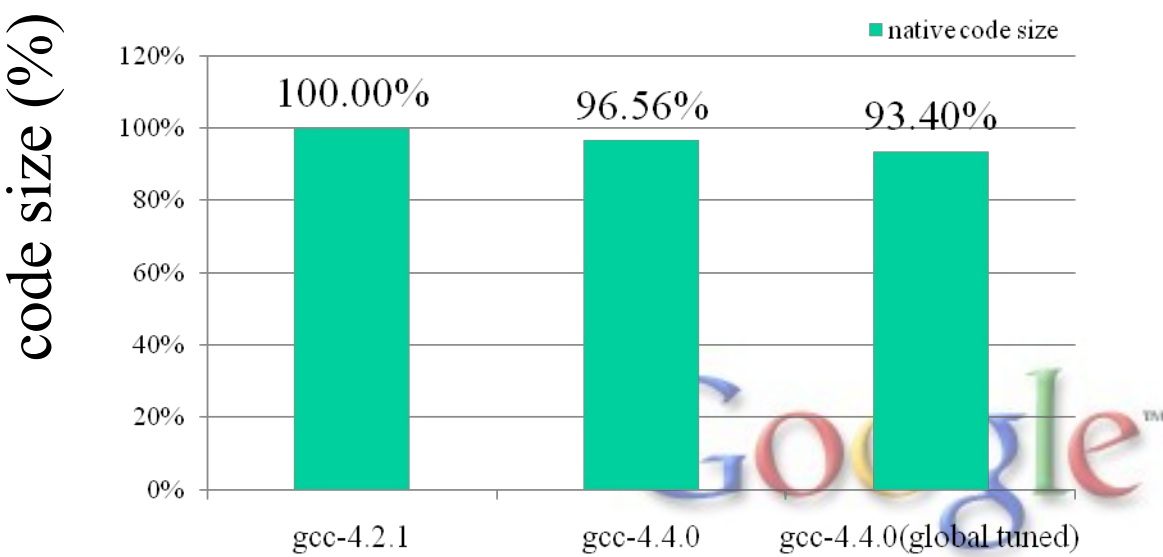
Reduce Code Size by Option Search

- We search for a configuration that reduces size the most using our compiler option search approach

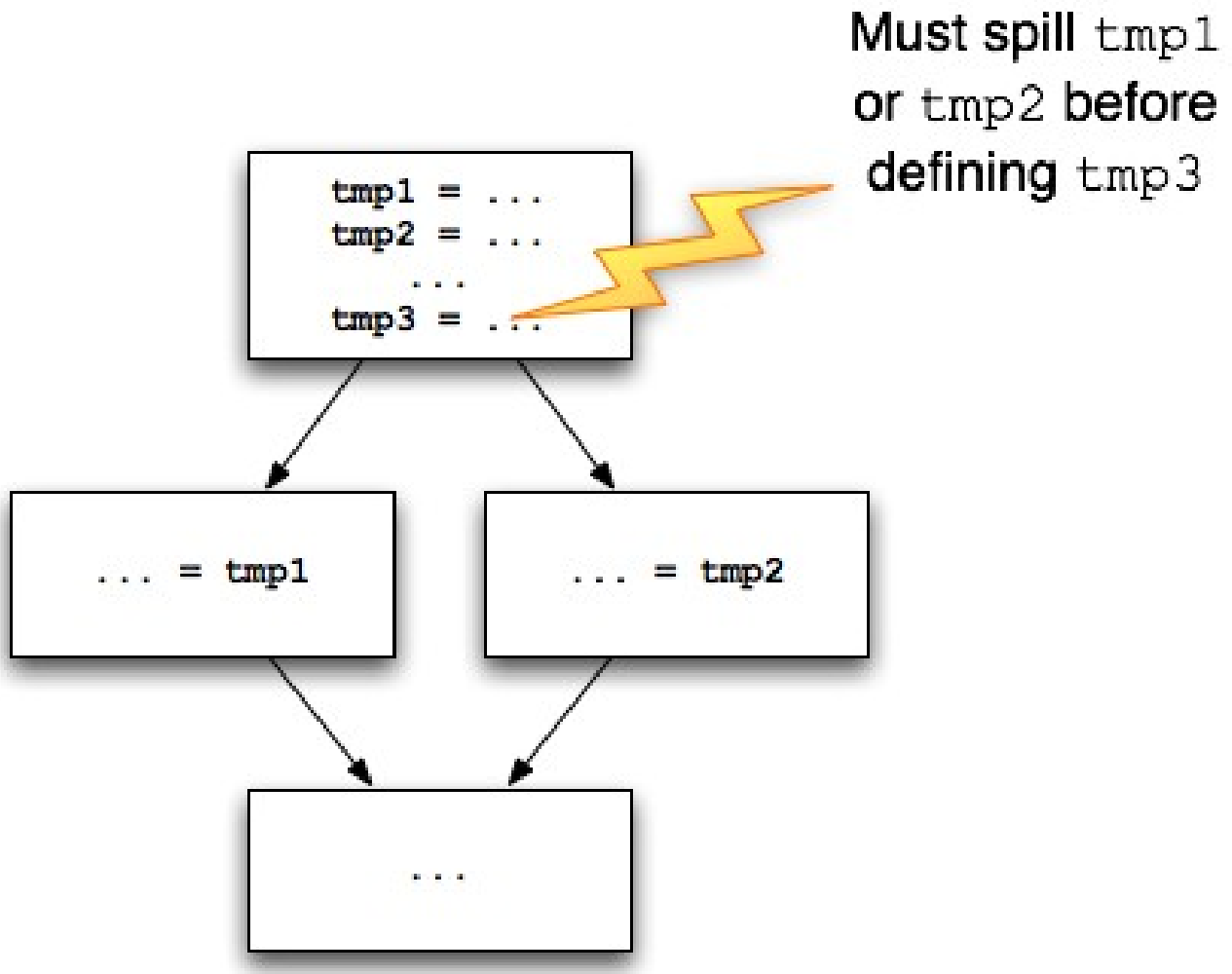
Android default inline options:
-finline-functions
-fno-inline-functions-called-once

Found options:
-finline
-fno-inline-functions
-finline-functions-called-once
--param max-inline-insns-auto=62
--param inline-unit-growth=0
--param large-unit-insns=0
--param inline-call-cost=4

	GCC-4.2.1	**GCC-4.4.0**	**GCC-4.4.0** **(tuned inline options)**
native system image	23,931,314	23,108,942	22,351,142

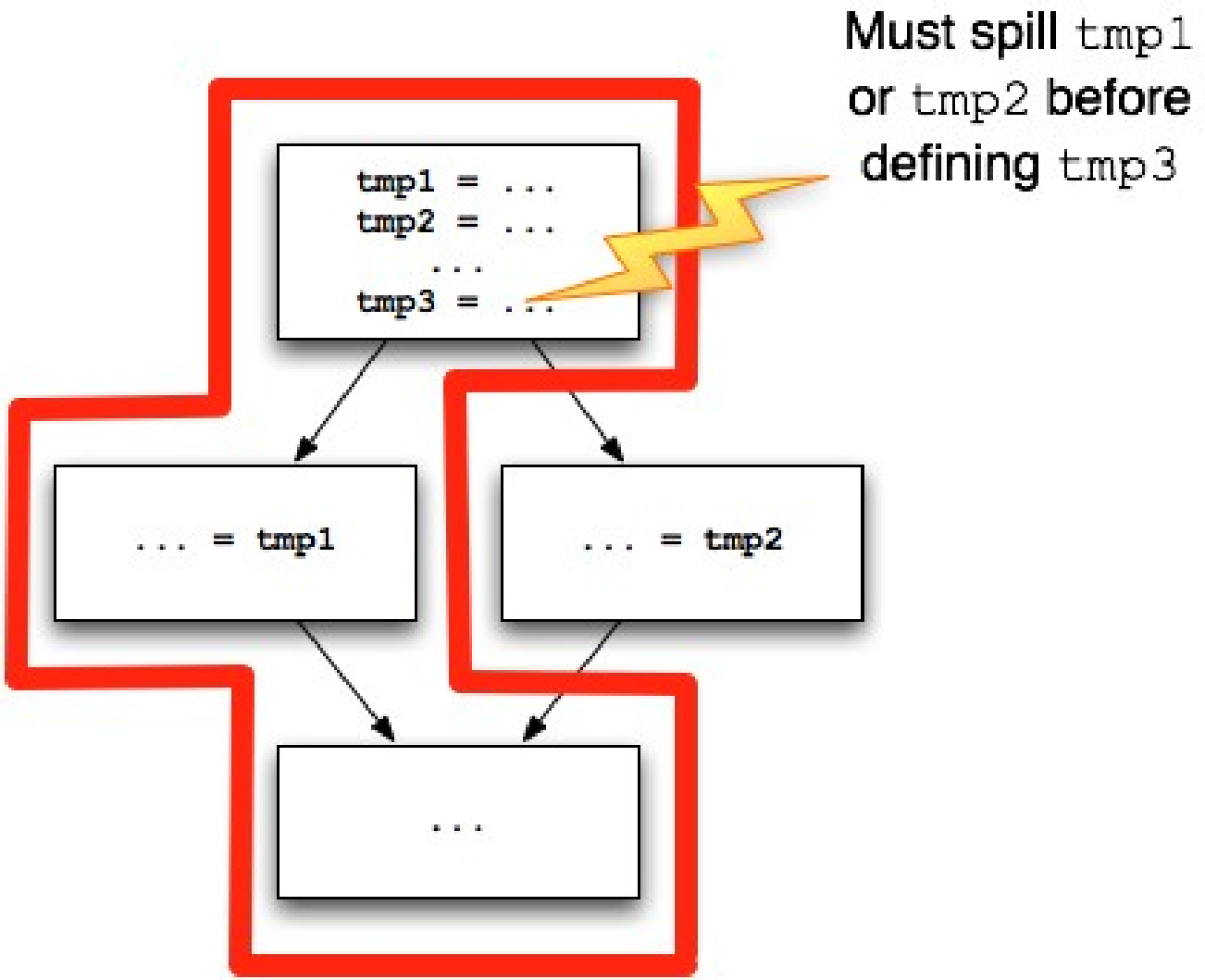


4. Profile-Guided Optimization: Our toolchain enables FDO (Feedback-Directed Optimization)



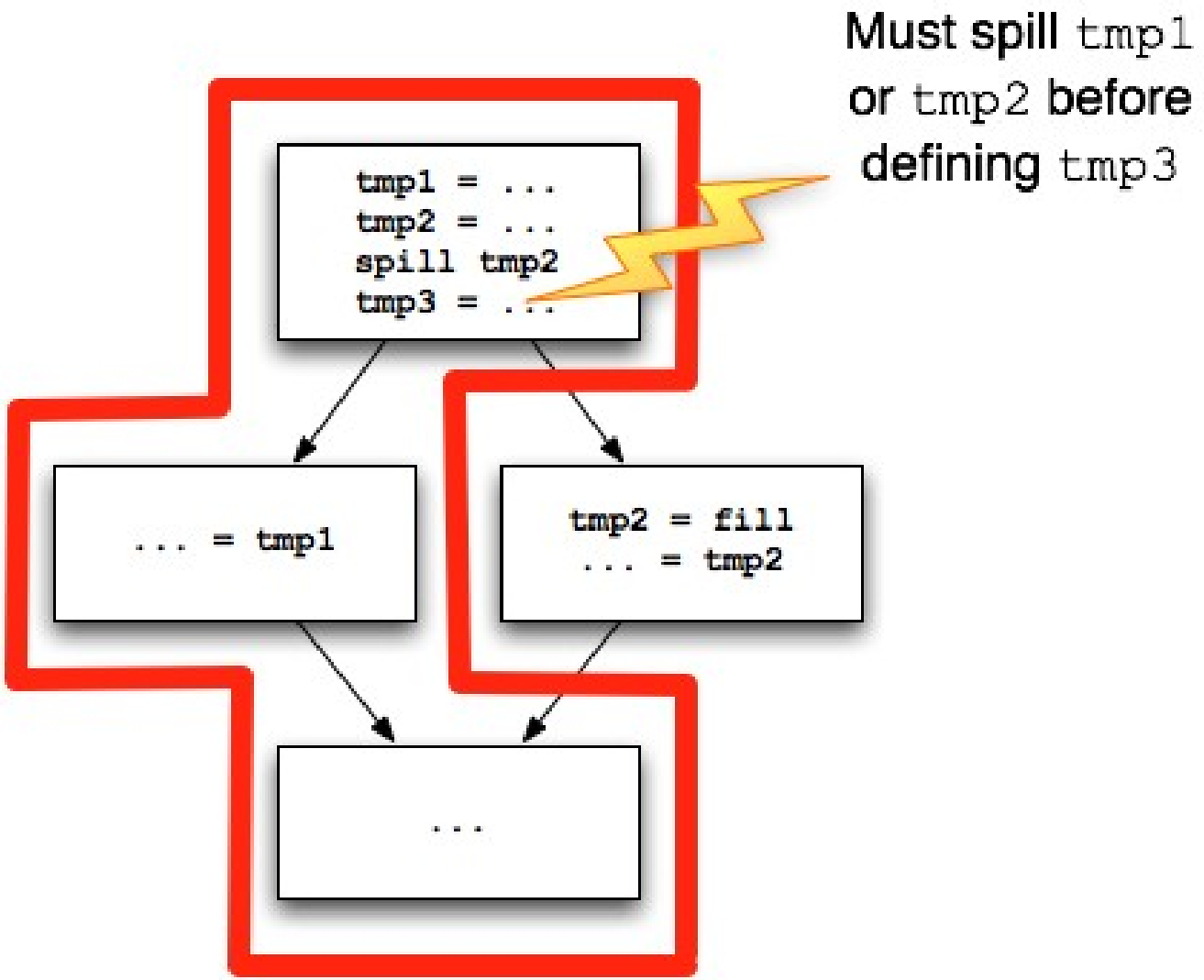
Google™

FDO Illustration

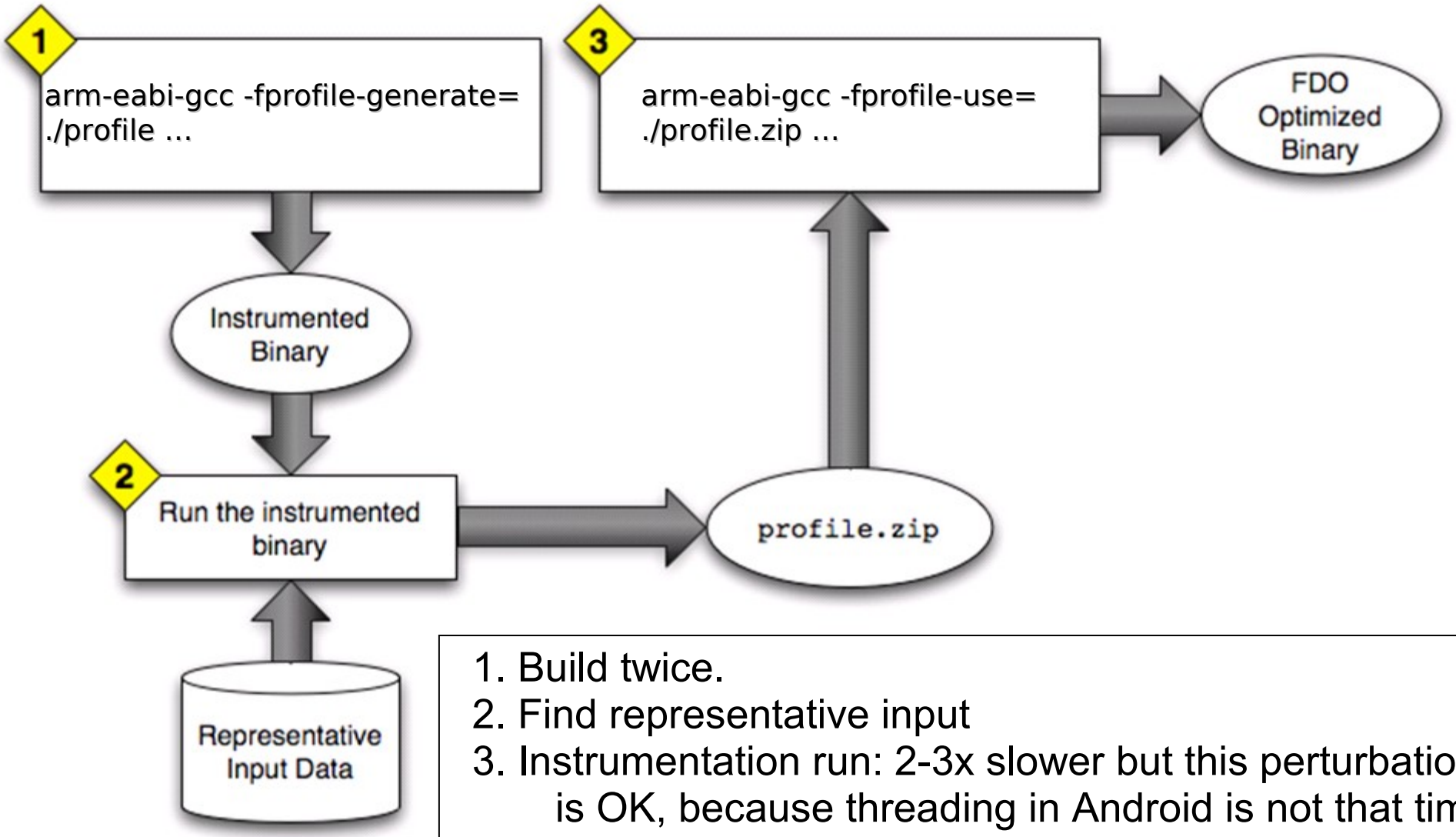


Google™

FDO Illustration



Instrumentation Based FDO



1. Build twice.
2. Find representative input
3. Instrumentation run: 2-3x slower but this perturbation is OK, because threading in Android is not that time sensitive (After all, 1 ARM11 or Coretex A8 core)
4. 1 profile per file, dumped at app exit.

Google™

FDO Performance

- Global hotness for ARM (HOT_BB_COUNT_FRACTION): 1% improvement on android's libskia (See below); smaller effects on smaller android benchmarks.

	default	fdo-default	fdo-modified
size of libskia	7879646	7396032	7319668
size reduction	0.00%	6.14%	7.11%
avg runtime	11.51	11.72	11.89
stdev (over 100 runs)	0.28	0.63	0.26
speedup	1	0.98	0.97

5. Scope-Enhancing Optimization
Inter-Procedural Optmizations (IPO)

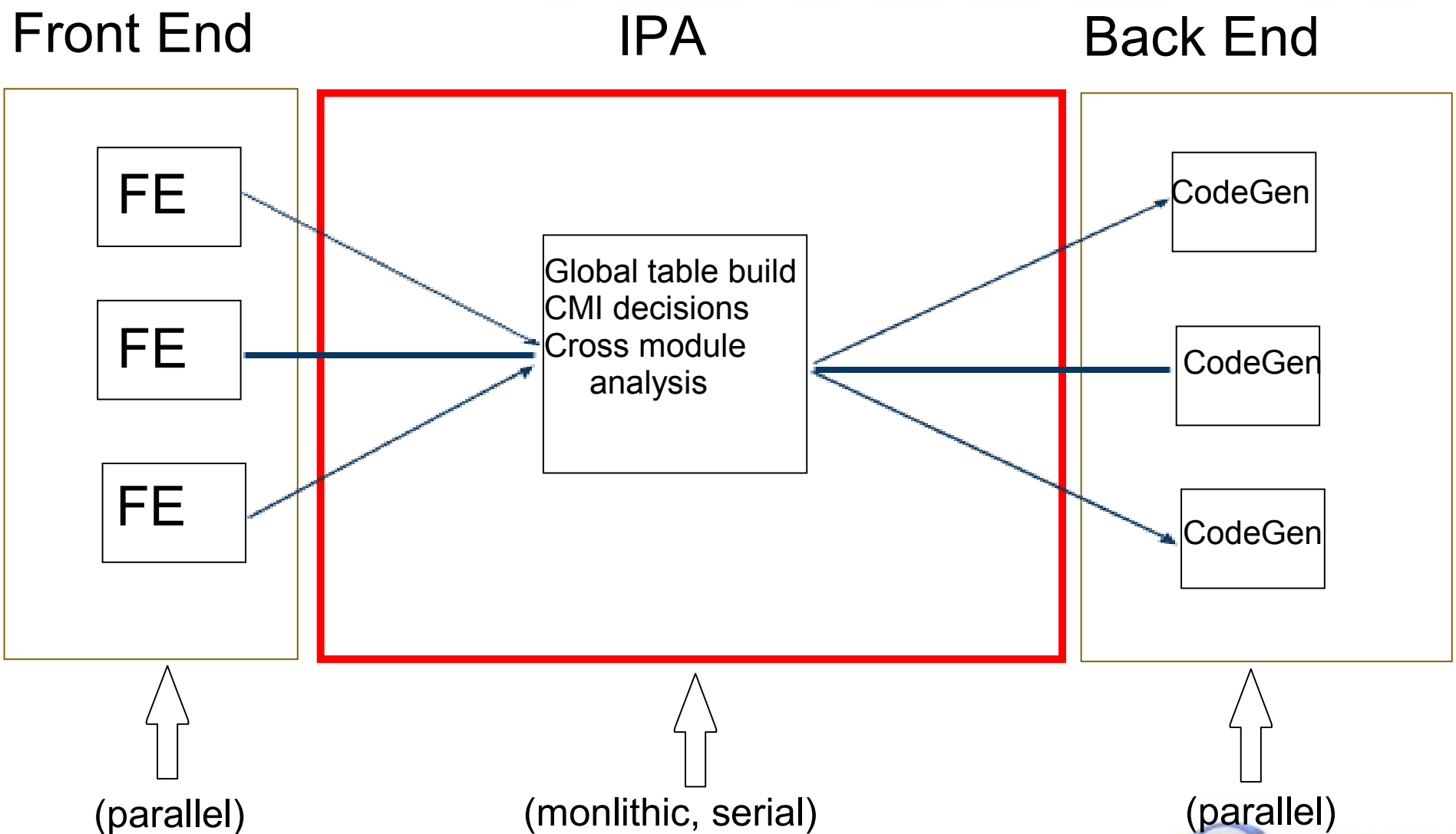
a.c:
```
int foo(int i, int j)
{
   return bar (i,j) + bar (j,i);
}
```

b.c:
```
int bar(int i, int j)
{
    return i - j;
}
```

- Optimization opportunity: Decided by scope of the code compiler can see
  - Scope limited mainly by artificial source boundaries

→ IPO enhances the scope

Problem with traditional IPO



Solution: Lightweight IPO (LIPO)

- To get the best potential out of IPO → Integrate IPO with FDO, seamlessly!
  - perf (IPO + FDO) > perf (IPO) + perf (FDO)
- Move IP analysis (IPA) to the end of training run execution,
  into the binary -- make global decisions earlier!
- Write IPA results into profile
- During **profile-use** compilation,
  - compile each file, as usual, with augmented profile
  - read additional IPA results
  - suck in auxiliary modules and extend scope

Google™

LIPO Improves Performance: Use -fripa

- LIPO targets C/C++: Android uses C/C++ (except for some assembly code)
- Baseline: FDO enabled
- Degradations are in noise range. Other benchmarks: Flat

- We just got the ARM version of LIPO to work:
  - Run:
    arm-eabi-gcc –fprofile-generate=/data/local/profile -fripa -mandroid
  - Replace:
    –fprofile-generate with –fprofile-use at the end of optimizaion

ARM results coming soon.

SPEC2000 on x86	Improvement	SPEC2006 on x86	Improvement
177.mesa	1.33%	433.milc	1.75%
164.gzip	3.83%	477.dealII	2.03%
175.vpr	3.43%	453.povray	12.74%
253.perlbmk	1.94%	445.gobmk	1.26%
254.gap	3.76%	458.sjeng	5.45%
255.vortex	21.12%	464.h264ref	8.51%
252.eon	3.42%	473.astar	0.72%

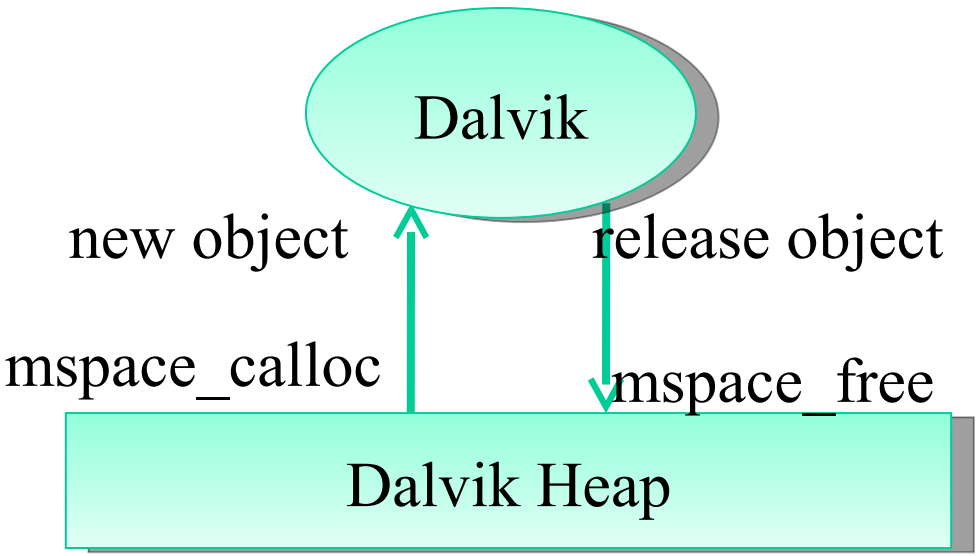
6. Redundancy Elimination: Identical Comdat (or Code) Folding (ICF)

- Identify identical functions and merge them at link time.
- Implemented in the binutils gold linker.
  - Triggered with option --icf.
- Debug support available through call tables.
- ICF on gold yields 5% on x86-64 binaries
- We are still getting gold linker to work with Android ARM. We estimate ~5% further Android size reduction on top of garbage collection. Stay tuned.

Google™

7. Optimizing Memory Management

- Each Dalvik Virtual Machine has its own heap
- Dalvik use dlmalloc API to manage its heap
  - Allocate memory by mspace_calloc
  - Release memory by mspace_free



Google™

Various Headrooms for Memory Management Optimizations

- Programs usually allocate/release objects frequently
  - Some of them have the same size

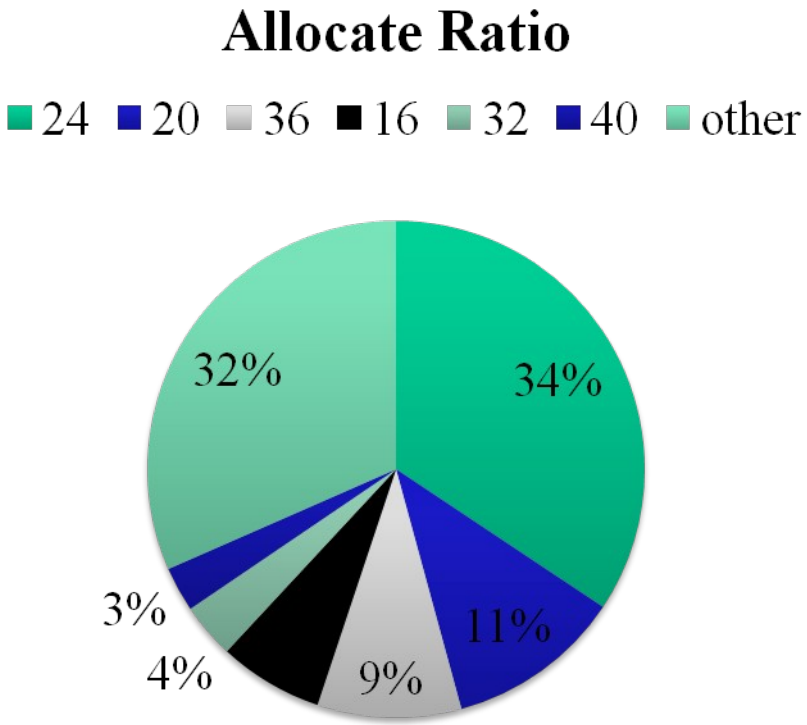
```
…
[Ljava/util/HashMap$Entry;:24
Ljava/util/HashMap$Entry;:24
Landroid/webkit/PerfChecker;:16
Landroid/webkit/LoadListener;:156
Landroid/webkit/ByteArrayBuilder;:20
Ljava/util/LinkedList;:20
Ljava/util/LinkedList$Link;:20
Ljava/util/LinkedList;:20
Ljava/util/LinkedList$Link;:20
Ljava/lang/String;:24
Ljava/lang/String;:24
Landroid/webkit/FrameLoader;:48
Ljava/lang/String;:24
…
```

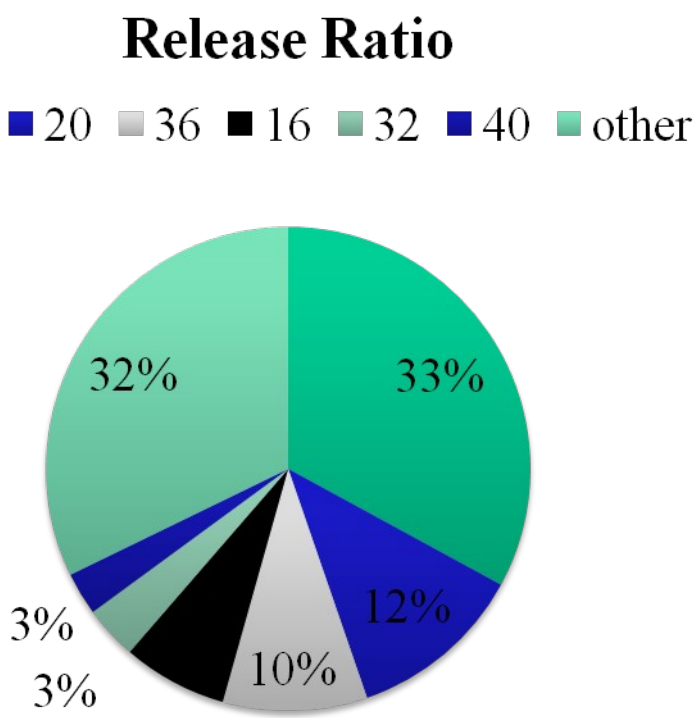
Size	Count	Ratio
24	16435	34.40%
20	5464	11.40%
36	4474	9.40%
…	…	…

Objects allocation log in WebViewBench

High ratio object sizes in WebViewBench

Observation of WebView Bench

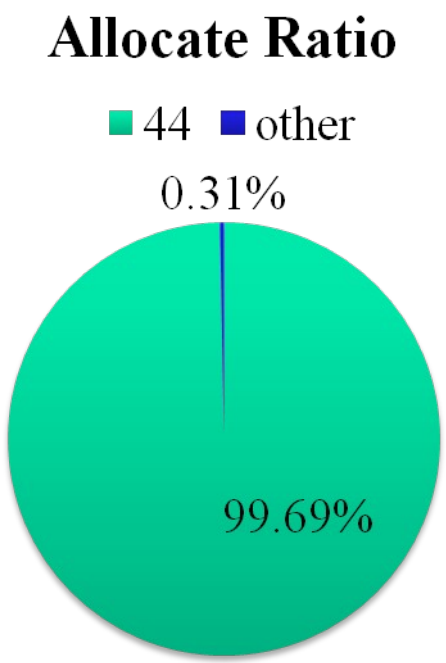


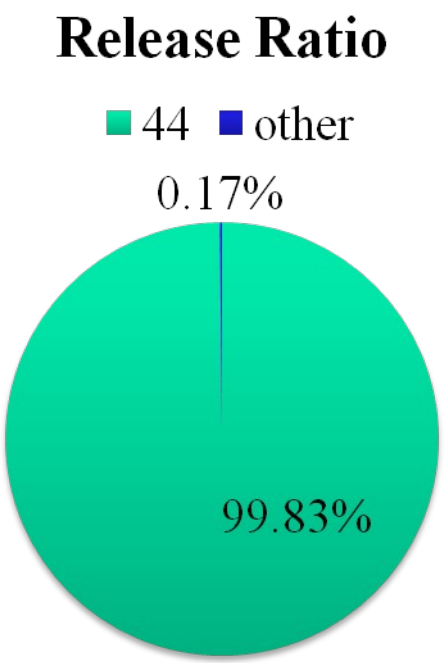


The size ratio between allocation and release is almost same

Google™

Observation of Fhourstones



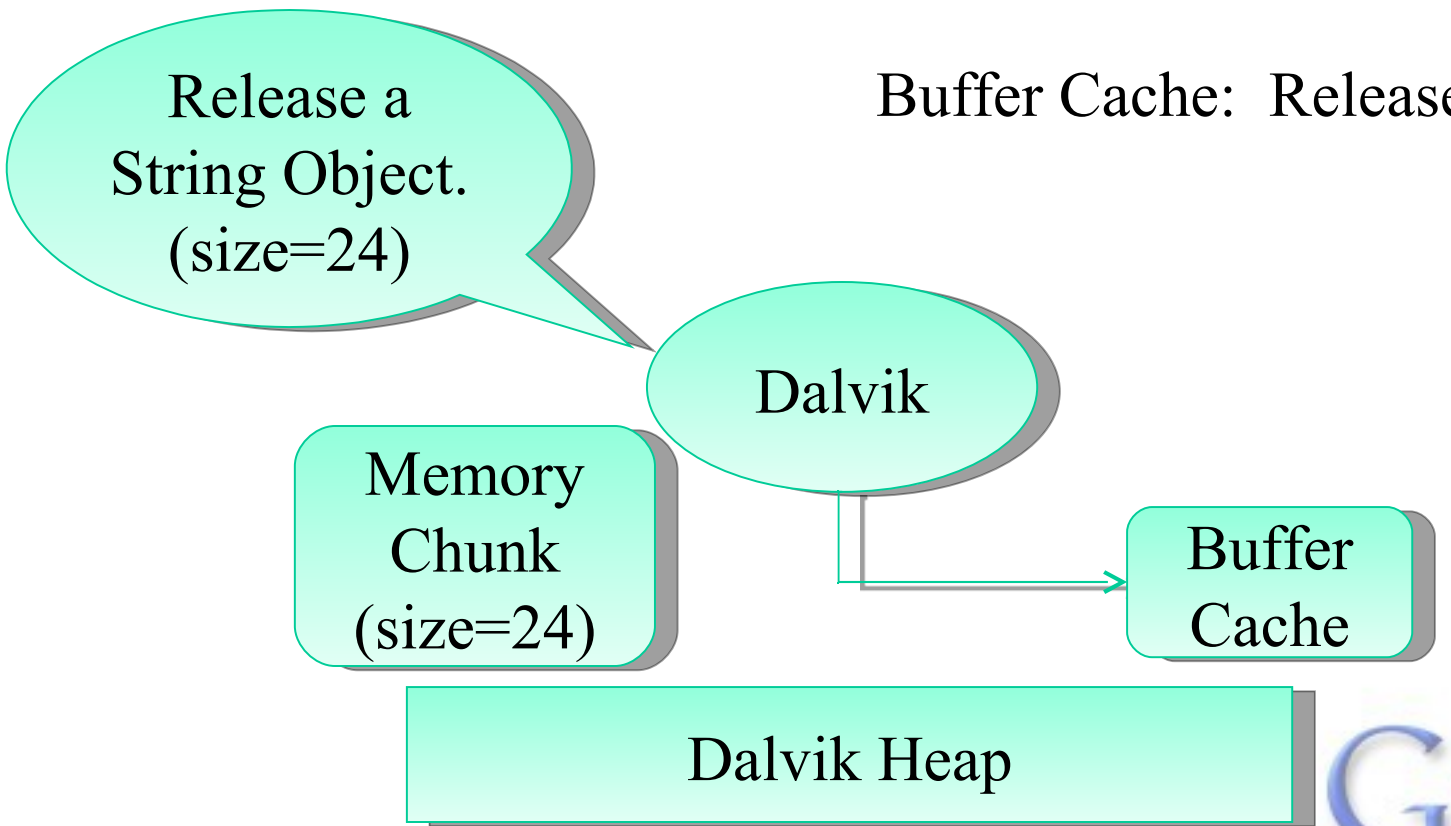


Ratio of Size = 44 is ***extremely high*** in this case

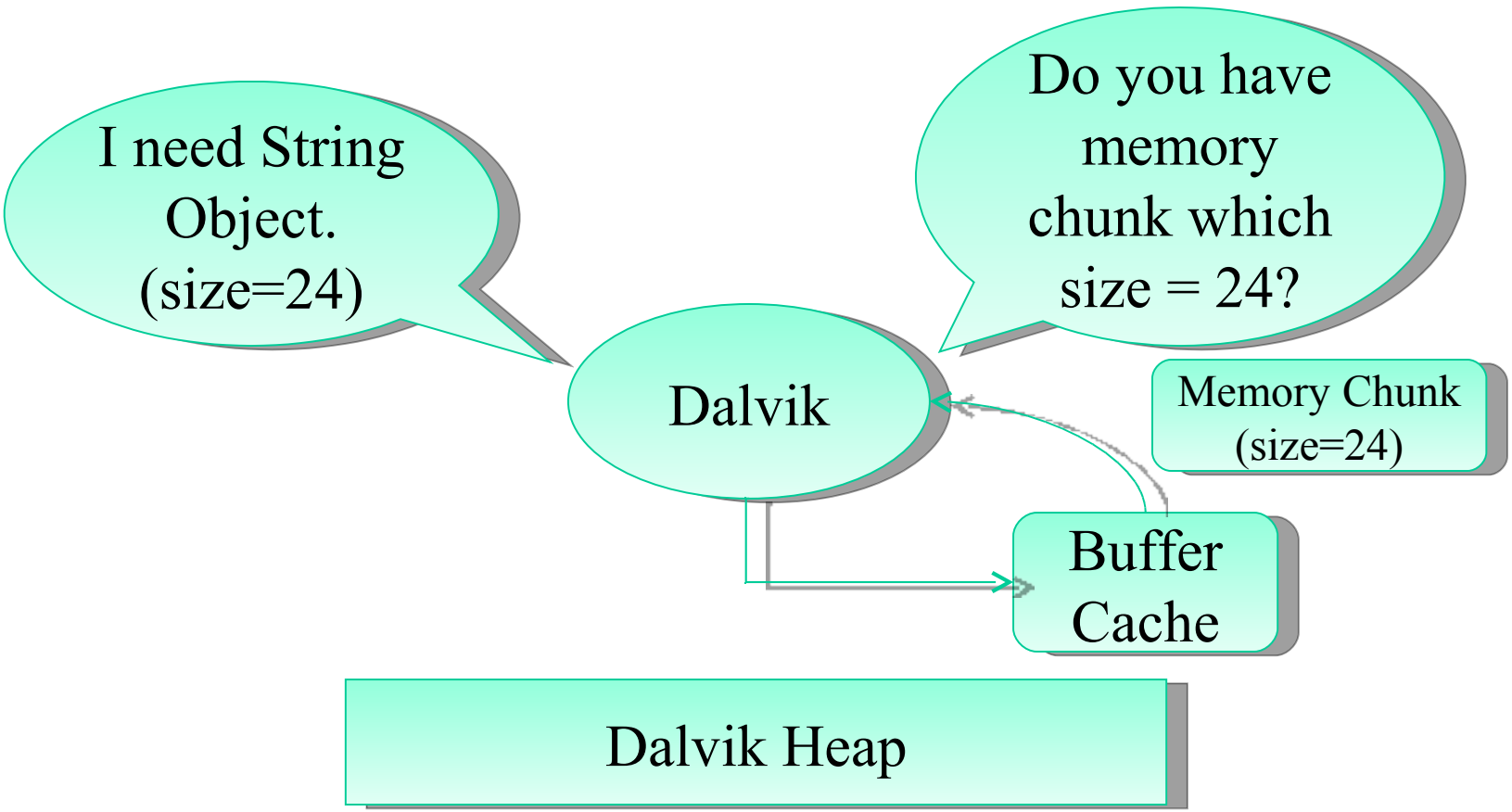
Google™

Many Objects Alloc/Released in Short Time

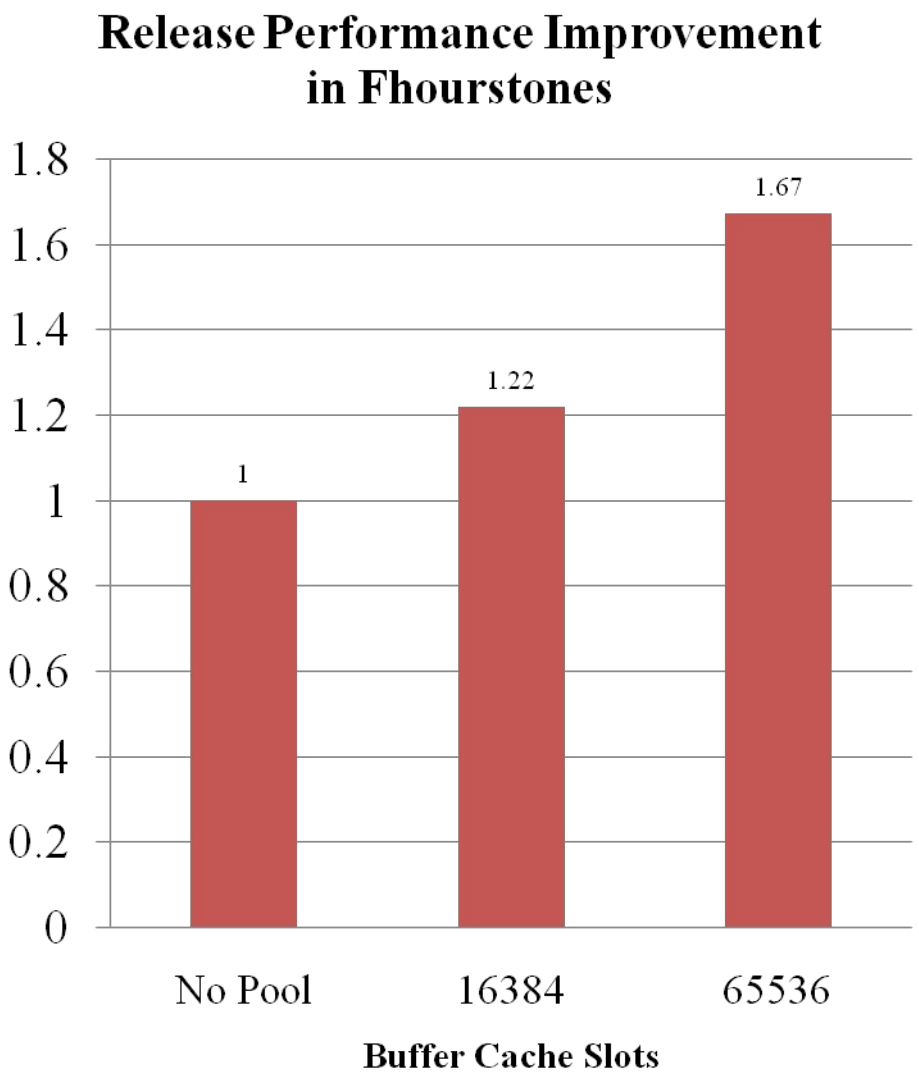
- Optimization: Add a buffer cache of memory chunks

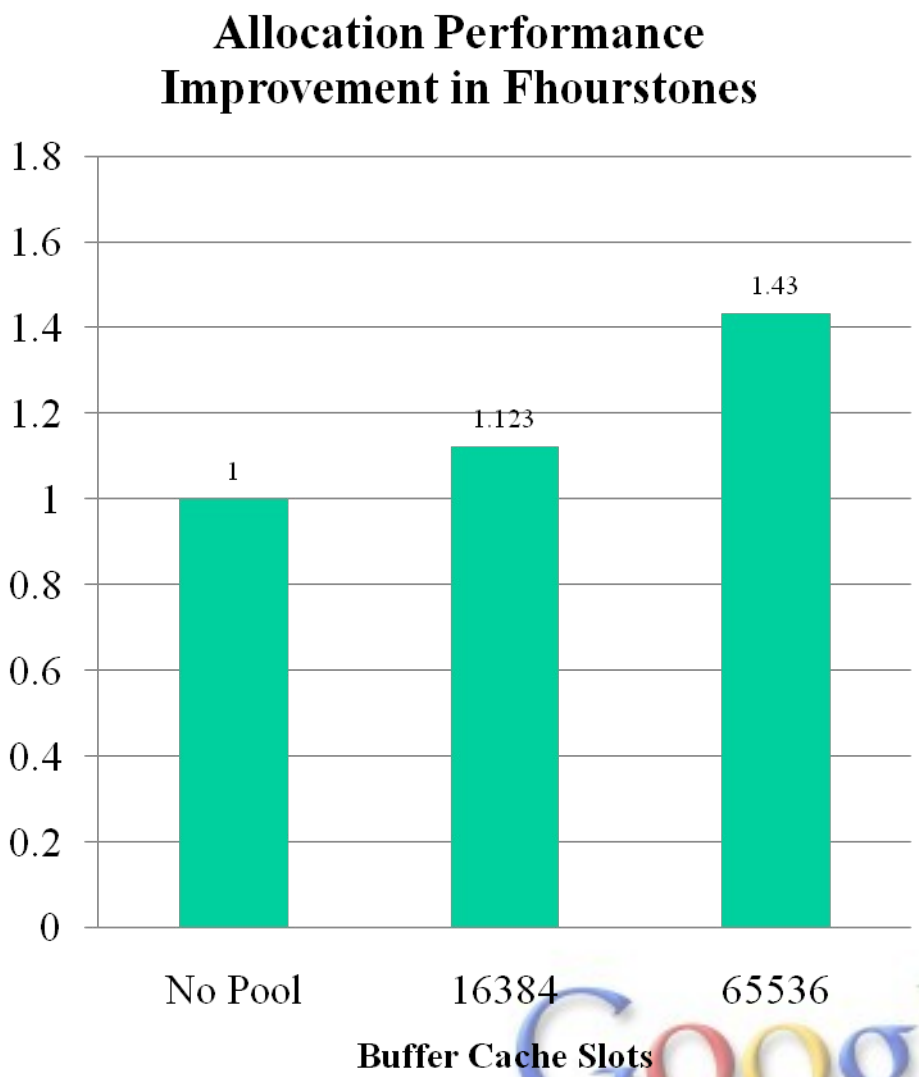


Buffer Cache: Allocate



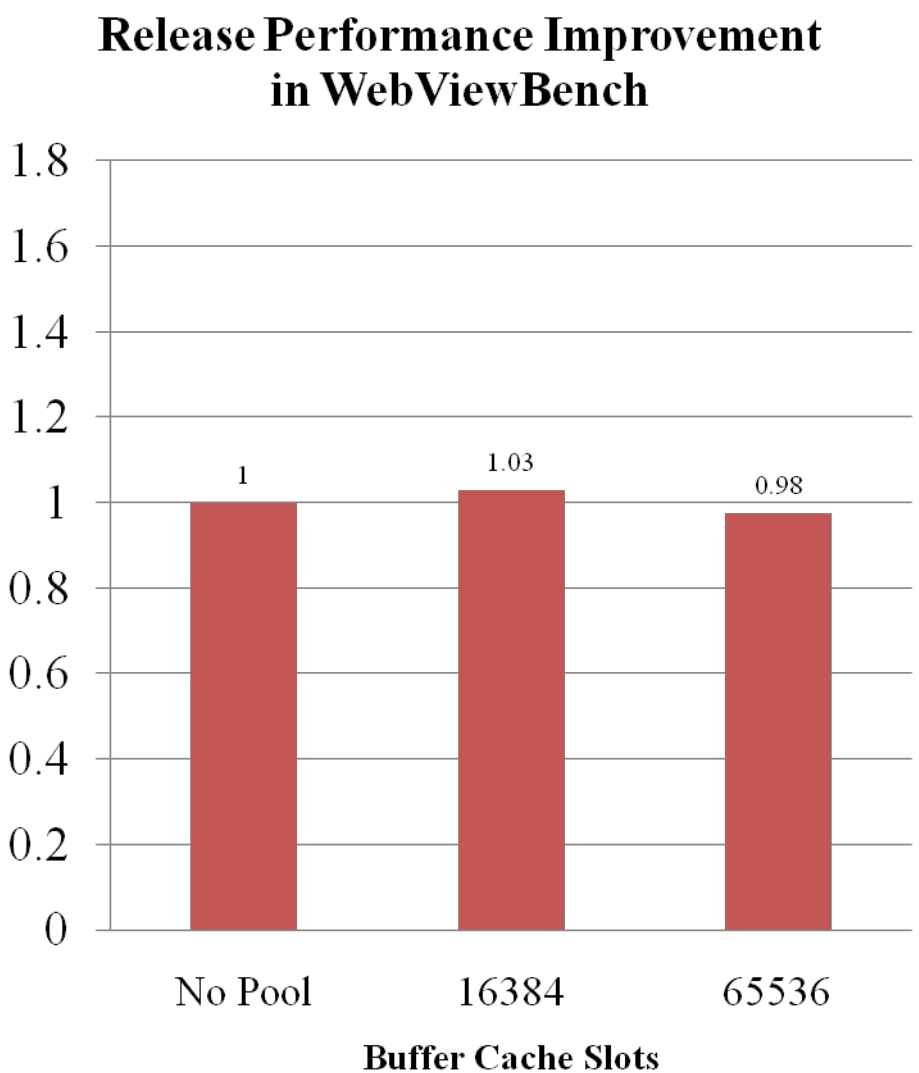
Experimental Result

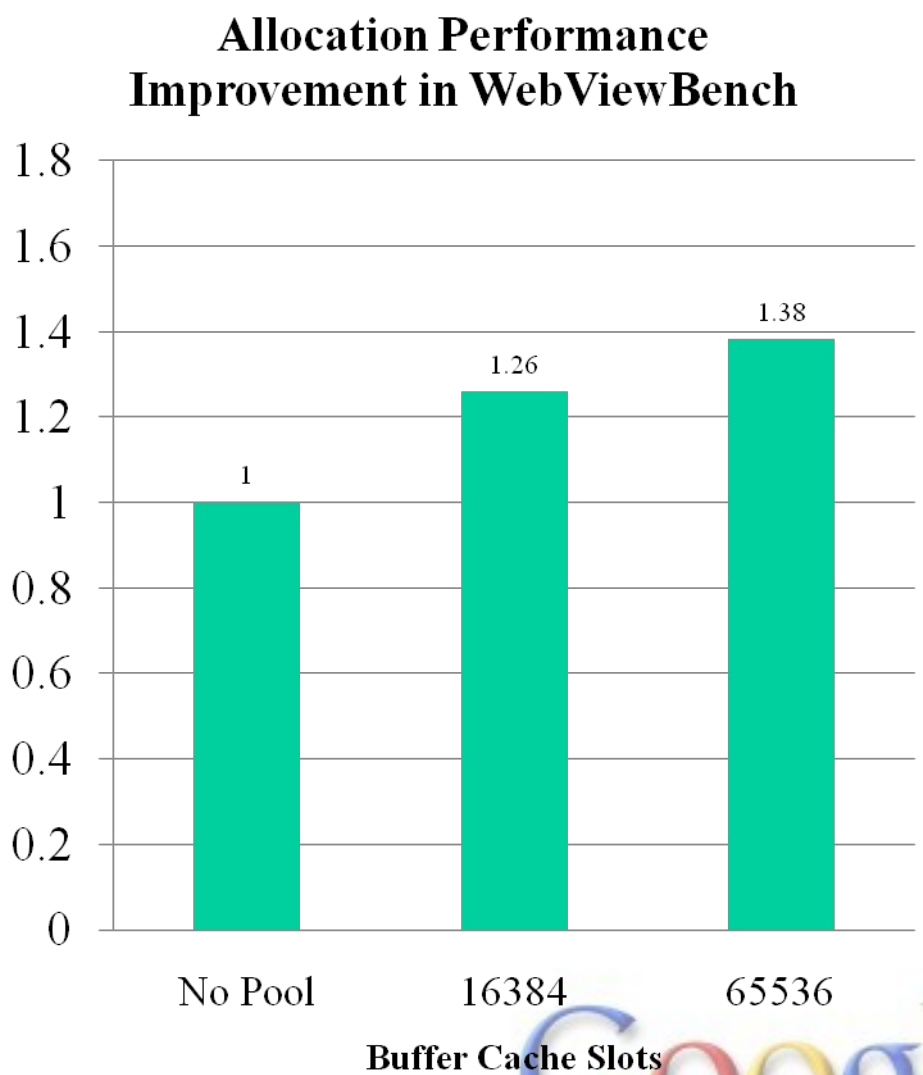




Google™

Experimental Result





Google™

Summary: Systematic Optimizations

- Toolchain: Regularly evaluate and leverage
  - E.g., leverage the newest lightweight IPO and ICF
- There is no substitute for thoughtful abstraction & specifications
- Systematic parameter setting: A key driver to performance
- Data-driven: Profile it
- Optimizing memory time for Android/Dalvik is important.

Google™