Google's C/C++ toolchain for smart handheld devices

Doug Kwan, Jing Yu and Bhaskar Janakiraman

Google Inc.
1600 Amphitheatre Parkway,
Mountain View, CA, USA
Email: {dougkwan,jingyu,bjanakiraman}@google.com

***Abstract*—Smart handheld devices are ubiquitous today and software plays an important role on them. Therefore a compiler and related tools can improve devices by generating efficient, compact and secure code. In this paper, we share our experience of applying various compilation techniques at Google to improve software running on smart handheld devices, using our mobile platforms as examples. At Google we use the GNU toolchain for generating code on different platforms and for conducting compiler research and development. We have developed new techniques, added features and functionality in the GNU tools. Some of these results are now used for smart handheld devices.**

I. Introduction

Smart handheld devices are ubiquitous today. Even in their relatively small form factors, they have enough computational power for applications like HD video decoding, video editing, face recognition, panorama photo stitching and more. Software running on some of these devices is equally powerful and complex. For example, the parts of Android® [1] Open Source Project [1] version 4.0.1 written in C, C++ and assembly languages alone are about 545 megabytes in size. Since software is a big component of a smart handheld device, we may want to pay attention to both its quality and its development process.

For smart handheld devices, there are ways to improve quality of software. A compiler toolchain can contribute to quality of software on devices. Compiler optimizations [2], [22] traditionally have been used to improve execution speed of software though they can also be used to optimize code size and power consumption [9], [17], [19].

Another key toolchain contribution is in the area of security. Attackers exploit poorly written code by injecting malicious code into the address space of a program. Many of these attacks can be discovered and stopped by using toolchain features [27] that generate code with runtime checks.

In addition, a toolchain can also improve software development process. For example, an efficient toolchain can improve productivity of a programmer by reducing the length of the edit/compile/debug cycle [26].

[1]Android and Chrome OS are registered trademarks of Google Inc. ARM and Thumb are registered trademarks of ARM Limited. Linux is a registered trademark of Linus Torvalds. All other brands or product names are the property of their respective holders.

978-1-4673-0269-2/12/$31.00 © 2012 IEEE

The benefits above are by no means exhaustive and their usefulness depends on the particular smart handheld device in question. For instance, on a device whose power consumption is dominated by its display, power optimization in the toolchain is of very little value. Likewise, on a device with large code storage, code compaction may irrelevant.

This paper is written for readers who are interested in smart handheld devices and with little or no background in toolchains. We give a general introduction to toolchains for smart mobile devices, using Google's mobile C/C++ toolchain as examples when appropriate. We discuss a number of topics, but not in great detail. Instead, we want this gentle introduction to spark readers' interests and to inspire them to explore on their own, using this paper as a starting point. Compiler optimizations for Android has been discussed previously in [19] but we cover more than one Google platform. We also give updates on some topics discussed in [19].

The organization of the paper is as follows. We first present some background information about toolchains and their uses on devices in section II. Then in sections III, IV, and V, we focus on a number of areas where a C/C++ toolchain improves software running on devices. Finally, we conclude the paper in section VI. This is followed by a list of references.

II. C/C++ Toolchain

In this section, we give some background information about C/C++ toolchains. We focus on these languages because they are commonly used in system programming on smart handheld devices.

A. GNU Compiler Collection

Google uses the GNU Compiler Collection (or GCC) [11] toolchain for some mobile platforms running on smart handheld devices. The GNU toolchain is a good choice for us because it supports many architectures commonly used on these devices and its source code is freely available. It allows Google to fix bugs quicker and to customize the toolchain to suit Google's needs. While we mainly discuss the GNU toolchain here, many things are not GCC specific and some are applicable to other open-source [20], [23] and commerical tools.

B. Maintaining An Open-Source Toolchain

Smart handheld device builders who use open-source tools may need to modify these tools locally because standard versions do not meet their needs. Under such circumstances, it would be better for them to maintain their own toolchains. This may sound a bit intimidating for some but there are resources available [11], [20].

Maintaining a customized open-source toolchain requires some planning and effort. Tools may have their own development and release cycles and device builders maintaining their own tools may be required to port changes to the tools they use for bug fixes and features, typically from versions of tools newer than theirs. Sometimes such *back porting* can be problematic, especially if there is a big gap in versions. To minimize back porting problems, it would be better to use reasonably up-to-date tools. This means tools need updating, which creates another problem since changes need to be *forward ported* to new versions but that may not be easy especially for big changes.

C. Contributing And Sharing Code

To avoid forward porting problems, local changes should be minimized. A good way to do this is to make changes useful to others and contribute these back to the open-source community. Google has been submitting back changes to GNU tools for years. In addition to bug fixes, we have contributed to various GNU tools projects like [6], [21], [25], [26], [28]. Sharing changes is good because Google can avoid maintaining these indefinitely and the community benefits from our work. Once changes are accepted by outside maintainers, forward porting is no longer required. Changes visible to outside are also less likely to be broken by others. To improve submission of changes, Google has set up GCC branches in FSF's repository and these are open to public [12].

Despite the need for planning and effort mentioned above, maintaining a customized open-source toolchain provides a good degree of flexibility. Smart handheld device builders should seriously consider the cost and benefit of this approach.

III. Performance

In this section, we would like to discuss how a toolchain can improve runtime performance of software running on a smart handheld device. We discuss performance first because it is the most visible benefit a toolchain can bring.

A. Measuring Performance

Before we can improve performance, we want to quantify it so that we know if performance is improved and by how much. We quantify generated code quality by metrics such as execution time of a benchmark. As programs running on smart handheld devices are usually different from those running on platforms like servers, we want to use benchmarks targeting these devices. For instance, Google developed and released different benchmarks [14], [15] for platforms running on smart handheld device. These benchmarks have the following properties:

- *Representative*: Each benchmark was carefully chosen to represent normal smart handheld device usage. Some benchmarks were extracted from source trees of Google's mobile platforms. Some benchmarks came from open-sourced games and tools, such as cximage, gnugo, etc.
- *Repeatable*: Timing, networking and other factors were removed from benchmark programs to reduce variation. Input sets were carefully chosen to reduce the impact of errors.
- *Isolated*: To get a stable base for measurement, one snapshot was taken from a stable release of a mobile platform. Benchmark programs and their dependent libraries were extracted from the snapshot. Each benchmark can be built and tested independent of others.
- *Relevant*: Some applications on our mobile platforms are not written in C/C++. To focus on the native part, native drivers were developed to drive native shared libraries for benchmarks extracted from our mobile platforms. A benchmark harness was designed to support various compiler experiments for a mobile toolchain.

B. Choosing Optimization Options

Choosing the best compiler options or even a good set of options to maximize performance is difficult due to size of the search space. There has been research in automatic selection of compiler options [5], [29]. In practice, it may be simpler to just use a simple optimization option, like *-O2* in GCC and a few hand-picked options for fine tuning.

Care must also be taken when using optimizations that may change results. For example, floating point optimizations like GCC's *-ffast-math* option [24] can cause differences in results. That may be acceptable for some applications, but not in general. Certain optimization options may also break non-conforming code like *-fstrict-aliasing* in GCC.

C. Architecture Specific Optimizations

Sometimes, it is possible to improve performance by using architectural features. For instance, many architectures support vector instructions, which can speed up certain computations significantly. We can use compiler options to enable architecture specific optimizations. A mobile platform may run on devices with different hardware features. In that case, we can compile the same source code to a number of binaries optimized for different variants. In some applications, multiple versions of the whole program are not desirable. We can compile only performance critical code for different variants and compile the rest for the lowest denominator. Hardware features can be selected at runtime. In this way, we may still need to use multiple versions of some parts of a program, like architecture-dependent shared libraries.

D. Feedback Directed Optimization

There are some optimizations that take more work than adding an option to use but they can bring significant benefits. A good example is *feedback-directed optimization* (FDO). It is a well-known technique supported by many compilers. To use FDO, we first build an instrumented binary, run the

instrumented binary with a representative workload to collect information about runtime behaviour and use the information as feedback to the compiler when recompiling the program. The recompiled program is optimized for the workload.

E. Link-Time Optimizations

A compiler typically processes one source file at a time. This limits the scope in which the compiler can optimize a program because it cannot see interaction between source files. GCC, LLVM and a number of other compilers support link-time optimization (LTO) [21], which allows a compiler to optimize a complete program [6]. LTO helps *Interprocedural Optimizations* (IPO) [22]. One example is *cross-module inlining* [18], in which inlining is done across module boundaries. LTO helps because a compiler can see beyond module boundaries. The compiler can do an even better job if LTO and FDO are used together.

F. Lightweight Interprocedural Optimization

Cross-module inlining across a big program has huge memory and processing overhead in the compiler. To address this, Google developed Lightweight IPO (LIPO) [18], which performs certain interprocedural optimizations (IPO) [22] without doing global analysis in the compiler. LIPO achieves this by shifting some of the analysis from compile time to runtime of the FDO feedback collection stage. A LIPO instrumented binary produces some information that would be produced by a global analysis.

IV. Binary Size

For smart handheld devices, especially those with limited storage, size of a program is as important as its performance. Instead of speed, a toolchain can also optimize for space. Space saving allows device builders to put more functionality within a given storage size or to use storage components of smaller capacities. In this section, we discuss binary size reduction.

A. Tweaking For Size

Usually compilers have options to reduce binary size. In GCC, we can use *-Os* to enable a combination of options that reduce binary size. Just like tuning options for speed, we can also use individual optimization options to fine tune the size of generated code. On some architectures, there are also architecture specific options that can be used to reduce binary size. For example, GCC can choose between using ARM®, Thumb® and Thumb-2 instruction sets for ARM CPUs to tune for different size requirements.

B. Speed/Size Trade-off

There is generally a trade-off between performance and size. Certain speed optimizations like inlining and loop unrolling [2], [22] can increase binary size. For size alone, we may want to disable such optimizations. As this may also decrease performance, we need to decide where we strike the balance. For example, Google tuned default inlining parameters of gcc-4.4.3 for ARM CPUs. We used benchmarks to test a number of inlining parameter combinations and picked the best one that satisfied our performance and size requirements.

C. Link-Time Garbage Collection

Both the GNU ld and the gold linker [26] support link-time garbage collection (GC), which identifies unreachable functions and data during linking and discards them. For this to be effective in ld and gold, source code must be compiled using GCC options *-ffunction-sections* and *-fdata-sections* [24]. Garbage collection is most effective when the number of global function and variable symbols are minimized, so authors of dynamic libraries should limit the visibility of symbols to match the need. Besides GC, there are also other good reasons to do so [10].

D. Identical Code Folding

Google developed a size optimization called Identical Code Folding (ICF) [25], in which the linker finds functions with same bodies and folds them into one copy. Some C++ ABIs [4], [8] use different constructors and destructors of a class in different contexts even though they are usually identical. Some compilers use function aliases to avoid duplication but GCC (up to version 4.6.2) emits identical copies in many cases. ICF removes these duplicated code and more.

In table I below, we show the effects of garbage collection (GC) and ICF on all ELF files in the /system directory of Android® Open Source Project version 4.0.1 configured for the *full_eng* target. We measured file sizes and instruction sizes in bytes. The latter were approximated by using the *size* command on Linux® to count total sizes of *.text* sections in files. We measured two ICF modes. In the *safe* mode, ICF only removes duplicated code if it is safe to do so. In the *all* mode, ICF removes all duplicated code even if it may break pointer comparisons. Google uses garbage collection and *safe* ICF on its mobile platforms.

optimizations	file sizes	%	insn. sizes	%
none	44702614		37839485	
GC	43819679	-1.98%	36988882	-2.25%
GC+ICF(safe)	42869285	-4.10%	36038488	-4.76%
GC+ICF(all)	42766003	-4.33%	35935210	-5.03%

TABLE I
Effects of GC and ICF on binary size

V. Security

We have just shown how to optimize code for speed and size, now we would like to discuss how a toolchain can improve security of software, using Chrome OS® as an example. The techniques used are not Chrome OS specific and can be applied to many other platforms as well.

A. Guarding Return Addresses On Stacks

Stack smashing [3] is a common attack, it involves overflowing an input buffer (an array) on a stack. Attacks may inject code on the stack but code-injection is not necessary for a successful attack [7]. A compiler can add runtime checks to

catch buffer overflows [27] but it is not necessary to check all array writes to improve security. Instead we can put a canary value before the return address to see if it has been overwritten when a function returns. Since there is only one check, the overhead is much smaller than checking all array writes. Note that just checking return addresses does not catch all similar exploits. Other attacks are possible like overwriting a function pointer on the stack.

B. Making Stack Not Executable

If a return address is overwritten to point to injected code on the stack, we can still stop an attack. It is sufficient to remove execution permission there. The GNU toolchain marks objects that need an executable stack. The GNU ld linker has an *execstack* option to handle this marking. For compatibility, we cannot turn off execution on stack if objects are not marked. The linker flag *–warn-execstack* can be used to detect missing markers, or, since need for an executable stack is rare, we may just whitelist binaries that need it.

C. Using Position-Independent Binaries

It is possible to attack a system without injecting code on the stack. An attacker can craft a sequence of return addresses to existing code locations, like in libraries [7]. To foil these attempts, we load binaries at random locations every time. This requires binaries to be position-independent. Some binaries like ELF shared libraries are already position independent. Others like executables load at fixed starting addresses, which may allow an attacker to find useful code snippets for exploits. We can build all binaries to be position-independent to improve security using GCC's *-fPIC*, *-fpic*, *-fPIE* and *-fpie* options.

D. Hardening Chrome OS®

Using techniques described above, Google hardened [16] Chrome OS against stack smashing. We followed [13] to pick security related compiler options for Chrome OS to improve security. Google uses GCC's stack smashing protector [27] to catch stack smashing in runtime. We also use the compiler option *-D_FORTIFY_SOURCE=2* to enable static buffer overflow checks in GCC. Dynamic executables in Chrome OS are position-independent to prevent attacks based on return-oriented-programming [7]. The overhead of hardening varies depending on the program, but is generally between 0% and 15%. On our internal benchmarks, we see average overhead of about 10%, with most of it coming from stack protection.

VI. Conclusion

Compiler optimizations and other code generation techniques can improve quality of software running on smart handheld devices. Google's use of the GNU toolchain for mobile platforms shows that a toolchain can help improving performance, size and security of software on devices.

Acknowledgment

The authors would like to thank organizers of VLSI-DAT 2012 conference for the invitation, internal reviewers at Google for their patience and valuable feedback, and finally many coworkers at Google for their contributions to our toolchain and related tools.

References

[1] Android Open Source Project, at http://source.android.com.
[2] Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd Edition.
[3] Aleph One, *Smashing the stack for fun and profit*, Phrack Magazine, Vol. 7, No. 49. (1996).
[4] ARM, *C++ ABI for the ARM Architecture*, ARM Document Number IHI 0041C, October 2009
[5] Guy Bashkansky and Yaakov Yaar, *Black Box Approach for Selecting Optimization*, SMART07 Workshop Proceedings, 2007, pp. 24-42.
[6] Preston Briggs, Doug Evans, Brian Grant, Robert Hundt et al.,*WHOPR - Fast and Scalable Whole Program Optimizations in GCC*, available at http://gcc.gnu.org/projects/lto/whopr.pdf.
[7] Erik Buchanan, Ryan Roemer and Stefan Savage, *Return-Oriented Programming: Exploits Without Code Injection*, Black Hat USA 2008 Briefings, Aug. 2008.
[8] Code Sourcery, *Itanium C++ ABI*, available at http://sourcery.mentor.com/public/cxx-abi/abi.html.
[9] Saumya K. Debray, William Evans, Robert Muth, Bjorn De Sutter, *Compiler Techniques for Code Compaction*, ACM Transactions on Programming Languages and Systems, Volume 22, Number 2, March 2000, pp. 378-415.
[10] Ulrich Drepper, *How To Write Shared Libraries*, Proceedings of UKUUG Linux Developers' Conference, 2002.
[11] *GCC, the GNU Compiler Collection* at http://gcc.gnu.org.
[12] *GCC: Anonymous read-only SVN access*, at http://gcc.gnu.org/svn.html.
[13] Gentoo Linux, *The Gentoo Hardened Toolchain*, at http://www.gentoo.org/proj/en/hardened/hardened-toolchain.xml.
[14] *Page Cycler Tests*, at http://www.chromium.org/developers/testing/page-cyclers.
[15] *Android Toolchain Benchmarks*, source code at https://android.googlesource.com/toolchain/benchmark.git.
[16] *System Hardening Features*, at http://code.google.com/p/chromium-os/wiki/SystemHardeningFeatures.
[17] M. Kandemir , N. Vijaykrishnan , M. J. Irwin and W. Ye, *Influence of Compiler Optimizations on System Power*, In Design Automation Conference 2002, pp. 304-307.
[18] David X. Li, Raksit Ashok and Robert Hundt, *Lightweight Feedback-Directed Cross-Module Optimization*, Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (2010), pp. 53-61.
[19] Shi-Wei Liao, *Smaller and Faster Android: Optimization and Toolchain Perspective*, COSCUP 2009. Slides and video at http://coscup.org/2009/en/program.
[20] *The LLVM Compiler Infrastructure*, at http://llvm.org.
[21] *Link Time Optimization*, GCC Wiki, at http://gcc.gnu.org/wiki/LinkTimeOptimization.
[22] Steven Muchnick, *Advanced Compiler Design and Implementation*.
[23] *Open64 Compiler*, at http://www.open64.net.
[24] Richard Stallman and the GCC Developer Community, *Using the GNU Compiler Collection*, for GCC version 4.6.2.
[25] Sriraman Tallam, Cary Coutant, Ian L. Taylor, David X. Li and Chris Demetriou, *Safe ICF: Pointer Safe and Unwinding aware Identical Code Folding in the Gold Linker*, Proceedings of the 2010 GCC Summit, pp. 107-114.
[26] Ian L. Taylor, *A New ELF Linker*, GCC Developers' Summit 2008, pp. 129-136.
[27] Perry Wagle and Crispin Cowan, *StackGuard: Simple Buffer Overflow Protection for GCC*, in proceedings of GCC Summit 2003, pp. 243-256.
[28] LeChun Wu and Jeffrey Yasskin, *Thread Safety Annotations and Analysis*, at http://gcc.gnu.org/wiki/ThreadSafetyAnnotation.
[29] Shengtong Zhong, Yang Shen and Fei Hao, *Tuning Compiler Optimization Options via Simulated Annealing*, in Proceedings of FITME '09, pp. 305-308.