

# Advanced Size Optimization of the Linux Kernel

Tim Bird  
Sony Mobile Communications

# Who am I?

- Tim Bird
- Researcher and open source guy at Sony
  - Recently switched to Sony Mobile Communications
- Researching system size for many years
- Background in extremely small systems
  - First programmed a TRS-80, 8k RAM
  - NetWare Lite – file and print server in 50K (1991)

# Outline

- The “size” problem
  - Characterization of bloat and strategy for dealing with it
- Automatic reductions
  - Constraint-based optimizations
- Additional kernel size research
- Resources for small-system work

# The Size problem

- System gains more features and bugfixes over time
  - Abstraction, layering, generalization all add to software
  - Result is system with lots of software that is never executed
- Bloat in open source
  - Generalization - Linux kernel supports everything from tiny sensors to supercomputers
  - Lots of features are configurable, but many are not

# Dealing with bloat

- Embedded devices have specialized use cases
  - Well, many do (TVs, cameras, set-tops, routers, etc.)
  - Most Android-based products are open platforms
    - Can run arbitrary software
- At a high level: Need to re-specialize the software for specific use cases unique to your product
  - But, want to continue to leverage open source community and software over time

# Bloat trajectory

- Software gets more generalized over time
  - Kernel growing by 10% per year for last 10 years
- Can't use strategy of manual tuning (i.e. config options)
  - 2.6.12 had 4,700 kernel config options, and 3.9 has about 13,000 options
  - An individual developer can't be an expert in so many different options
    - Manual configuration doesn't scale
- Need to rely on automated methods of reduction

# Bloat in kernel vs. user space

- In desktop or server, for user-space programs virtual memory makes bloat issue less important
  - Pages are loaded on demand – only the working set of the program is in memory
- For kernel, pages are always loaded
- Embedded devices often do not have swap

# Automatic Reduction (Intro)

- In order to reduce the software, it is necessary to distinguish used code from unused code
  - Without resorting to manual configuration
- This research includes a few different techniques for finding and eliminating unused code in the Linux kernel



# But first...The story of my own 8 bytes of bloat

- Not my only bloat addition, but this one really bugged me
- I added a conditional check in kdb
  - Found a bug in kdb, when a particular option was used in kdb startup file
  - I added a patch to fix problem, but now every kernel has this fix
  - Only 8 bytes, but this is how the kernel gets bigger over time
  - Very few people use kdb startup files, or that particular option
  - My own contribution of code and overhead, unneeded by almost everyone!
- More correct solution would be to detect condition at compile-time, and eliminate the runtime test, but this was impractical

# Generalizing the Problem of Bloat

- System doesn't know invariant system states (e.g. limitations on inputs to functions)
  - It's easy to omit a driver for unused or not-present hardware
  - It's difficult to omit code paths or error handling for inputs that will never occur due to fixed use cases
- Can we identify fixed inputs to kernel functions, and use compiler to optimize the code?

# Example of invariant state in embedded – uid

- There are uid references throughout the kernel
  - References in storage, file system, task structures, accounting
- uid is ultimately set by `setuid()`, by the 'login' program
  - Login does a lookup and validates user in `/etc/passwd`
- What if `/etc/passwd` only has 'root' and no others?
- `Setuid()` can only be called with a value of 0
- Can I encode this constraint on the system?

# Types of constraints

- Syscalls never called by any program
- Kernel command-line arguments never used
- Parameters that are never used, or limits on possible parameter values (setuid(uid))
- /proc or /sys values never referenced

# Auto-reduce project

- Find automated ways to reduce the kernel
  - Link-time optimization
  - System call elimination
  - Kernel command-line argument elimination
  - Kernel constraint system
- Additional research
  - Link-time re-writing
  - Cold-code compression

# Link-Time Optimization

---

# Link Time Optimization

- LTO is a new GNU toolchain feature (gcc 4.7+)
  - Save extra meta-data (gimple format) at compile-time
  - Use meta-data at link time to do whole-program optimization
- Obsoletes gcc -ffunction-sections
- Has slow link step, but much better code optimization
- See <http://lwn.net/Articles/512548>

# Link-Time Optimization

- Andi Kleen created patches to support this compiler option for the Linux kernel
  - Patches are for Intel architecture
  - See <http://lwn.net/Articles/512548/>
  - Code available at: <git://github.com/andikleen/linux-misc>
- I did a few patches for ARM architecture
- Requires gcc 4.7 and linux-binutils 2.22.51.0.1 or later



# LTO Benefits

- Opens up a whole new class of optimizations
- Performance improvements: (very preliminary results)
  - (x86) Hackbench – 5%, network benchmark – up to 18%
- Size improvement:
  - (x86) No size improvement reported by Andi
  - (ARM) 6% kernel size reduction (384K in my testing)

# Link-Time Optimization Results

- I demoed first LTO kernel running on ARM at ELC 2013 (February 2013)
  - World's first, that I know of!!
  - TI panda board, mem=24M
  - 384K smaller kernel image



Kernel	non-LTO	LTO
Compile time	1m 58s	3m 22s
Image size	5.85M	5.46M
Meminfo Total	17804K	18188K

# LTO Problems

- Longer build times
  - Link takes about 1.5 minutes, for small kernel config
- More memory required for builds
  - 9G for x86 allyesconfig
- Andi found a few subtle bugs from optimizations
  - E.g. Duplicate code elimination caused a pointer comparison failure
  - These should be eliminated with newer toolchain versions

# Possible Future Benefits of LTO

- Can automatically drop unused code and data
  - Maybe reduce ifdefs in kernel
- Partial inlining – e.g. only inline some code, like tests at beginning of functions
- Optimize arguments to global functions
  - Drop unnecessary args, optimize inputs/outputs, etc.
- Detect function side effects, and optimize caller
  - e.g. Keep values in registers over call

# Possible Future Benefits of LTO (cont.)

- Detect read-only variables and optimize
- Replace indirect calls with direct calls and optimize
- Do constant propagation, and function call specialization based on that
  - If a function is called commonly with a constant, make a special version of the function optimized for that
    - e.g. `kmalloc_GFP_KERNEL()`

# System Call Elimination

---

# System Call Elimination

- In theory, it's pretty simple:
  - Determine which syscalls are used by all user-space programs
  - Remove unused system calls from the kernel
- In practice, there are a few details to take care of...

# Finding used/unused system calls

- Initial test using a single binary
  - Statically linked busybox
- Scan object files (assembly) for specific syscall code sequences
- Program: find-syscalls.py
  - Produces a list of used and unused syscalls for an object file
  - Also, shows warnings for weird syscall code sequences



# Problem with dynamic linked libraries

- Libc includes calls to all syscalls
  - When libc is statically linked, functions are automatically eliminated if not referenced
  - That doesn't happen when libc is dynamically linked
- Need a mechanism to scan all binaries in system, and eliminate unreferenced functions from dynamic libs
  - Libopt – MontaVista program to remove unreferenced functions in libraries
- Note: must be re-run if new binaries are introduced to system
  - In practice, new binaries very rarely add new syscalls

# Eliminating syscalls in kernel

- Added mechanisms in kernel to remove unused syscalls
  - Added `UNUSED()` macro, which converts syscall reference from `sys_foo()` to `sys_ni_syscall()` - used in `arch/arm/kernel/calls.S`
  - Created `unused_syscall.h` file (initially empty)
  - Created `syscall_usage.h`, with per-syscall `asmlinkage` definitions
- Created `mark-unused.py` for saving syscall usage data in source
  - Adds macro `UNUSED()` around any unused syscalls in `calls.S`
  - Adds `is_unused_foo` definitions in `unused_syscall.h` file

# Asmlinkage details

- Syscalls are declared using SYSCALL\_DEFINE macros in `include/linux/syscall.h`
- By default, asmlinkage macro is defined with `__visible`, which becomes `__attribute__((externally_visible))`
  - This exists specifically to keep syscalls from disappearing during linking – but.. we *want* unused ones to disappear
- My mechanism declares asmlinkage without the `__visible` attribute, so that LTO can eliminate the syscall

# System Call Elimination Results

- Total number of syscalls: 395
- Syscalls marked unused: 211

Kernel	Syscalls removed	Size reduction
unoptimized	161	94,980
optimized	120	47,860

- Optimized kernel was hand-configured to remove unneeded features (49 CONFIG option changes)

# System Call Elimination Notes

- Finding syscalls in ARM is pretty reliable
  - Only 2 assembly sequences where manual evaluation was needed to determine call
- Finding syscalls may be hard on other architectures
  - Affected by method of syscall invocation, register usage, etc.
  - Could compile with no optimization just for syscall determination pass

# Kernel Command-line Argument Elimination

---

# Kernel Command-line Argument Elimination

- Kernel command-line args documented in Documentation/kernel-parameters.txt
- Defined with `__setup()` and `early_param()` macros from `include/linux/init.h`
  - Approximately 480 `__setup()` routines in kernel source
  - About 200 `__setup_*` in `System.map` on ARM kernel build (98 `__setup_str_*`)
  - About 230 `early_param` routines in kernel source

# Argument Elimination Mechanism

- Define new macros `__setup_used()` and `early_param_used()`
  - If `CONFIG_PARAM_USED_ONLY`, then make `__setup()` and `early_param()` definitions empty
- Create a list of used params (in constraint config)
- Change 'used' routines to use macros `__setup_used()` and `early_param_used()`



# Command-line Elimination Results

- Unoptimized kernel: 19K reduction

	base	test	bytes changed	percent
text:	7680084	7663472	-16612	0%
data:	362868	360516	-2352	0%
bss:	745312	745184	-128	0%
total:	8788264	8769172	-19092	0%

- Optimized kernel: 6K reduction

	base	test	bytes changed	percent
text:	1653672	1648920	-4752	0%
data:	131636	130244	-1392	-1%
bss:	50688	50528	-160	0%
total:	1835996	1829692	-6304	0%

# Kernel Constraint System

---

# Kernel Constraint System

- Goal is to find constraints that can be applied system-wide (both kernel and user-space), to optimize the software

# Background: Lessons from Linux-tiny

- Linux-tiny added lots of CONFIG options
  - Was hard to get accepted upstream
- After a few years of mainlining the big options, each additional option only resulted in small reductions (e.g. < 5k)
  - Not worth maintaining in isolation
  - Not enough incentive to accept into mainline
- Required too much knowledge to turn on/off configuration items
  - Most developers prefer to keep features in kernel unless they understand the full impact of removal
- Linux-tiny patches had to be maintained out-of-tree

# Constraint objectives

- Constraint represents a change that can result in optimization by the compiler
- Change can be automatically applied
- Robust across software versions
  - ie. the change is not in the form of patches
- Cross-layer (both kernel and user-space)
- Ability to generate new constraints from previous ones

# Mechanism

- Constraint configuration file
  - Declaration of constraint type
  - Information needed to apply the constraint
- Set of programs to modify the kernel source
  - Started with a mixture of automation and some manual steps
  - Targeted full automation
- Usually, replaced kernel source with comments
- Tools are run as a pre-build step
  - Requires integration with the build system

# Details for Auto-Reduce program

- Processes constraints.conf
  - Has a declaration for each constraint
- Modifies kernel source code, in a way that compiler can optimize
- Uses sub-module: ex. find\_refs.py
  - May compile kernel multiple times during application of constraints
    - It is NOT FAST

# Source modifications

- Easy to clear modifications with 'git checkout'
  - Should start with clean tree (to avoid losing local changes)
- Most common example is replacing code with constants
- Try to keep original code in comment, so altered code can be inspected (in case of bugs)



# Constraint types

- Structure field values with constant values
  - Allows removing the field from the structure
- Constant function argument values
- Limited set of values for function arguments
- Unused function calls
  - Unused syscalls already dealt with

# Constant structure field values

- This is the type for “uid==0”
- For this constraint type:
  - Remove field from structure
  - Locate all references to field
  - Replace each reference with constant (in source)

```

--- a/include/linux/cred.h
+++ b/include/linux/cred.h
@@ -123,5 +123,5 @@ struct cred {
 #define CRED_MAGIC_DEAD  0x44656144
 #endif
-    kuid_t      uid;          /* real UID of the task */
+//    kuid_t      uid;          /* real UID of the task */
     kgid_t      gid;          /* real GID of the task */
     kuid_t      suid;         /* saved UID of the task */

```

```

--- a/fs/fcntl.c
+++ b/fs/fcntl.c
@@ -208,5 +208,5 @@
     if (pid) {
         const struct cred *cred = current_cred();
-        filp->f_owner.uid = cred->uid;
+        filp->f_owner.uid = /*cred->uid*/ 0;
         filp->f_owner.euid = cred->euid;
     }

```

# Locating references

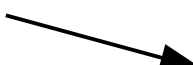
- “uid” is found in over 50 different structures in the kernel (file system, accounting, struct cred)
  - The one I mean to constrain is in struct cred, but “uid” appears in many others
- Over 2300 references to “uid” in kernel
- Lots of references to struct cred.uid via macros
  - Some without any “uid” in the macro name
- Too difficult to do simple grep for, or write a parser for

# Solution to Locating References

- Use the compiler
- Method:
  - Remove field from structure
  - Build the source code
  - Record errors for missing field
  - Use that to pinpoint line numbers
  - Parse line to find actual field reference to remove

# Problems with Using Compiler

- Only references that are built in the current configuration are detected
  - If configuration changes, then must re-run tool to reapply constraints
- If macros are used, pattern for line parse is not obvious
  - Must manually determine macros to modify – ugh!



```

--- a/include/linux/cred.h
+++ b/include/linux/cred.h
-@@ -342,5 +343,5 @@
  })

-#define current_uid()      (current_cred_xxx(uid))
+#define current_uid()      0
#define current_gid()      (current_cred_xxx(gid))
#define current_euid()      (current_cred_xxx(euid))

```

# Kernel Constraint results

- Uid==0 constraint yielded code savings of 304 bytes
  - Total of 45 changes made to code, including macro
- Only came up with 7 constraints, before termination of project
  - Total savings 2688 bytes
    - OK – that's pretty disappointing

# Constraints Discussion

- Had hoped that constraints would “cascade”
  - Uid is used in multiple data structures, and all others derived from this (well, kind of – fsuid)
  - But, making it constant at the source (cred->uid) did not propagate to other structures
  - Possibly due to aliasing
-

# Constraints Conclusion

- Uid==0 patch would never be accepted upstream
- I'm waiting a few kernel versions, to determine “robustness” of patch-averse approach
- If there were thousands of constraints that could be detected and applied automatically, the constraint system might work
  - Could gather constraints over time
- For now, constraint system is a failure...



# Additional Research

---

# Additional Research

- Some research that I found while investigating this
  - I did not conduct this research, but merely present it here for your consideration
    - Warning: it's a bit old (2.4.25)
  - Research done at University of Gent and University of Arizona
- Two areas:
  - Link-time re-writing
  - Cold-code compression

# Link-time Binary Re-Writing

---

# Link-time Binary Re-Writing

- Consists of a tool to examine the assembly code for a program (from the binary) and do extra analysis
- Finds common instruction sequences
- Does code reach-ability analysis from a whole-program graph
- May use original source code (1 did and 1 did not)
- Special techniques for finding indirect functions

# Link-time Re-Writing Issues

- Big problem is use of indirect pointers
  - Had to consider possible pointer assignments to get correct function reach-ability graph
  - Reduce set of functions that are assignable, to those whose address was taken somewhere in the program

# Link-time Re-Writing Results

- 23.23% code size reduction on 2.4.25 ARM kernel [Fe]
- About 12.6% image size reduction on 2.4.25 ARM kernel [Chanet]
  - About 186K

# Cold-code compression

---

# Cold-code Compression

- Mechanism to compress sections of the Linux kernel in RAM
- Identify “cold code” through profiling
- Store those sections compressed
- Uncompress at runtime if a section area is ever invoked



# Cold-code Compression Details

- NOT a virtual memory approach (no faults taken)
- Code is replaced with stubs, which uncompress and call real code when called
- Is one-way – code is never re-compressed
- Not all code will be uncompressed
  - Only some exceptional code, and never any unreachable code, will be decompressed

# Cold-code Compression Issues

- Division of code into frozen and non-frozen parts
  - Division of code into basic blocks, then instrumentation at runtime
  - Compressing only blocks that exceeded a size threshold
- Managing concurrency when decompressing code
  - Research implementation had fully re-entrant decompressor to avoid locking (and potential priority inversion)
  - Did a post-decompression check, to see if a block was decompressed twice (indicating a decompression race condition), and freed the second block

# Cold-code Compression Results

- Yielded a net 17.8% reduction in uncompressed image size
  - 2.4.25 ARM kernel
  - Including overhead of code for decompression mechanism and stubs
- Reduced from 1209K to 1029K in size
  - 180K net reduction

# Conclusions

- Significant size reductions are available using:
  - New compiler features (LTO)
  - Aggressive specialization
- Kernel constraint value awaits further research
- Research indicates that additional savings are possible using link-time re-writing or cold-code compression
  - These should be re-verified – may be partially obsoleted by LTO
- More work is needed to continue fighting kernel bloat

# Resources

---

# Tiny Distribution

- Poky-tiny distribution in the Yocto Project
- See <https://wiki.yoctoproject.org/wiki/Poky-Tiny>
- Good for testing and further research

# Papers

- Chanet D., De Sutter B., De Bus B., Van Put L., and De Bosschere K. 2007. Automated reduction of the memory footprint of the linux kernel. *ACM Transactions on Embedded Computer Systems Volume 6, 4, Article 23*
  - From Ghent University
- He H., Trimble, J., Perianayagam S., Debray S., Andrews G. 2007 “Code Compaction of an Operating System Kernel” *Proceedings of the International Symposium on Code Generation and Optimization*
  - From University of Arizona
  - Available at: <http://www.cs.arizona.edu/solar/papers/kernel-compaction.pdf>

# eLinux.org page

- I will try to continue collecting information at:
  - [http://elinux.org/System\\_Size\\_Auto-Reduction](http://elinux.org/System_Size_Auto-Reduction)
- This will include patches, yocto recipes, scripts and results



**Thanks for your time**

---

Questions??

My e-mail: [tim.bird\(at\)sonymobile.com](mailto:tim.bird(at)sonymobile.com)

---

**SONY**  
make.believe

"SONY" or "make.believe" is a registered trademark and/or trademark of Sony Corporation.  
Names of Sony products and services are the registered trademarks and/or trademarks of Sony Corporation or its Group companies.  
Other company names and product names are the registered trademarks and/or trademarks of the respective companies.