



Embedded Linux size reduction techniques

Michael Opdenacker
Bootlin *michael.opdenacker@bootlin.com*



- ▶ Michael Opdenacker
- ▶ Founder and Embedded Linux engineer at Bootlin
 - ▶ Embedded Linux **expertise**
 - ▶ **Development**, consulting and training
 - ▶ Strong open-source focus
- ▶ Long time interest in embedded Linux boot time, and one of its prerequisites: small system size.
- ▶ From **Orange**, France



Penguin from Justin Ternet

(<https://openclipart.org/detail/182875/penguin>)



Why reduce size?

There are multiple reasons for having a small kernel and system

- ▶ Run on very small systems (IoT)
- ▶ Run Linux as a bootloader
- ▶ Boot faster (for example on FPGAs)
- ▶ Reduce power consumption
Even conceivable to run the whole system in CPU internal RAM or cache (DRAM is power hungry and needs refreshing)
- ▶ Security: reduce the attack surface
- ▶ Cloud workloads: optimize instances for size and boot time.
- ▶ Spare as much RAM as possible for applications and maximizing performance.

See https://tiny.wiki.kernel.org/use_cases



Reasons for this talk

- ▶ No talk about size since ELCE 2015
- ▶ Some projects stalled (Linux tinification, LLVM Linux...)
- ▶ Opportunity to have a look at solutions I didn't try: musl library, Toybox, gcc LTO, new gcc versions, compiling with Clang...
- ▶ Good to have a look again at that topic, and gather people who are still interested in size, to help them and to collect good ideas.
- ▶ Good to collect and share updated figures too.



How small can a normal Linux system be?

▶ RAM

- ▶ You need 2-6 MB of RAM for an embedded kernel
- ▶ Need at least 8-16 MB to leave enough space for user-space (if user-space is not too complex)
- ▶ More RAM helps with performance!

▶ Storage

- ▶ You need 2-4 MB of space for an embedded kernel
- ▶ User space can fit in a few hundreds of KB.
- ▶ With a not-too-complex user-space, 8-16 MB of storage can be sufficient.



Compiler optimizations

- ▶ `gcc` offers an easy-to-use `-Os` option for minimizing binary size.
- ▶ It is essentially the optimizations found in `-O2` without the ones that increase size

See <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html> for all available optimizations



Using a recent compiler

Compiling for ARM versatile, Linux 4.10

- ▶ With gcc 4.7: 407512 bytes (zImage)
- ▶ With gcc 6.2: 405968 bytes (zImage, -0.4%)

A minor gain!



Using gcc LTO optimizations

LTO: Link Time Optimizations

- ▶ Allows gcc to keep extra source information to make further optimizations at link time, linking multiple object files together. In particular, this allows to remove unused code.
- ▶ Even works with programs built from a single source file! Example: `oggenc` from <http://people.csail.mit.edu/smcc/projects/single-file-programs/oggenc.c> (1.7 MB!)
- ▶ How to compile with LTO:

```
gcc -Os -flto oggenc.c -lm
```

See again <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html> for details.



gcc LTO optimizations results

Compiling `oggenc.c`

- ▶ With gcc 6.2 for x86_64:
 - ▶ **Without LTO**: 2122624 bytes (unstripped), 1964432 bytes (stripped)
 - ▶ **With LTO**: 2064480 bytes (unstripped, -2.7%), 1915016 bytes (stripped, **-2.6%**)
- ▶ With gcc 6.2 for armelhf:
 - ▶ **Without LTO**: 1157588 bytes (unstripped), 1018972 bytes (stripped)
 - ▶ **With LTO**: 1118480 bytes (unstripped, -3.4%), 990248 bytes (stripped, **-2.8%**)

Note: the x86_64 size is not meant to be compared with arm code. 64 bit code is bigger than 32 bit code, that's expected.



gcc vs clang

Let's try to compile `oggenc.c` again:

- ▶ Compiled with **gcc** 6.2.0 on x86_64:
`gcc oggenc.c -lm -Os; strip a.out`
Size: 1964432 bytes
- ▶ Compiled with **clang** 3.8.1 on x86_64:
`clang oggenc.c -lm -Os; strip a.out`
Size: 1865592 bytes **(-5%)**
- ▶ gcc can catch up a little with the LTO option:
`gcc oggenc.c -lm -flto -Os; strip a.out`
Size: 1915016 bytes **(-2.7%)**

Note that gcc can win for very small programs (-1.2 % vs clang on `hello.c`).



ARM: arm vs thumb instruction sets

- ▶ In addition to the *arm 32 bit instruction set*, the ARM 32 bit architecture also offers the *Thumb instruction set*, which is supposed to be more compact.
- ▶ You can use `arm-linux-objdump -S` to distinguish between arm and thumb code.

Arm code

32 bit instructions

```
00011288 <main>:
11288: e92d4870    push    {r4, r5, r6, fp, lr}
1128c: e28db010    add     fp, sp, #16
11290: e24ddf61    sub     sp, sp, #388 ; 0x184
```

↑
Addresses
multiples of 4

Thumb code

16 bit instructions

```
00011288 <main>:
11288: b5f0       push    {r4, r5, r6, r7, lr}
1128a: b0e5       sub     sp, #404 ; 0x194
1128c: af06       add     r7, sp, #24
```

↑
Addresses
multiples of 2



ARM: arm vs thumb instruction sets (2)

- ▶ To compile in arm mode:

```
arm-linux-gnueabi-gcc -marm oggenc.c -lm
```

Result: 1323860 bytes

- ▶ To compile in thumb mode (default mode for my compiler!):

```
arm-linux-gnueabi-gcc -mthumb oggenc.c -lm
```

Result: 1233716 bytes (-6.8%)

- ▶ Notes:

- ▶ Thumb instructions are more compact but more are needed, which explains the limited size reduction.
- ▶ Thumb mode can be the default for your compiler!
- ▶ In my tests with `-marm`, the binary was a mix of Arm and Thumb code.



How to get a small kernel?

- ▶ Run `make tinyconfig` (since version 3.18)
- ▶ `make tinyconfig` is `make allnoconfig` plus configuration settings to reduce kernel size
- ▶ You will also need to add configuration settings to support your hardware and the system features you need.

`tinyconfig:`

```
$(Q)$(MAKE) -f $(srctree)/Makefile allnoconfig tiny.config
```



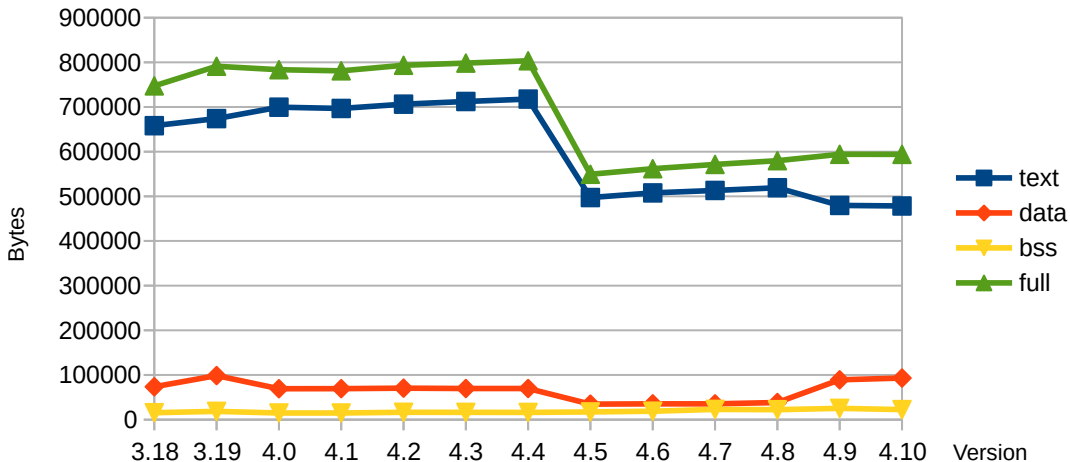
```
# CONFIG_CC_OPTIMIZE_FOR_PERFORMANCE is not set
CONFIG_CC_OPTIMIZE_FOR_SIZE=y
# CONFIG_KERNEL_GZIP is not set
# CONFIG_KERNEL_BZIP2 is not set
# CONFIG_KERNEL_LZMA is not set
CONFIG_KERNEL_XZ=y
# CONFIG_KERNEL_LZO is not set
# CONFIG_KERNEL_LZ4 is not set
CONFIG_OPTIMIZE_INLINING=y
# CONFIG_SLAB is not set
# CONFIG_SLUB is not set
CONFIG_SLOB=y
```



```
CONFIG_NOHIGHMEM=y  
# CONFIG_HIGHMEM4G is not set  
# CONFIG_HIGHMEM64G is not set
```

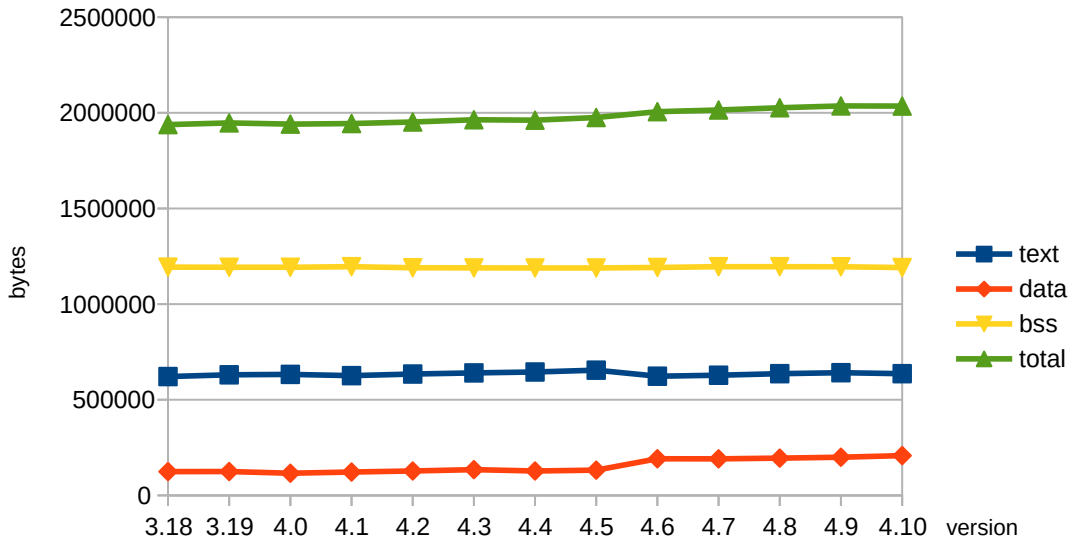


tinyconfig Linux kernel size (arm)





tinyconfig Linux kernel size (x86)





Linux kernel size notes

- ▶ We reported the `vmlinux` file size, to reflect the size that the kernel would use in RAM.
- ▶ However, the `vmlinux` file was not stripped in our experiments. You could get smaller results.
- ▶ On the other hand, the kernel will make allocations at runtime too. Counting on the stripped kernel size would be too optimistic.



Kernel size on a system that boots

Linux 4.10 booting on QEMU ARM VersatilePB

- ▶ `zImage`: 405472 bytes
- ▶ `text`: 972660
- ▶ `data`: 117292
- ▶ `bss`: 22312
- ▶ `total`: 1112264

Minimum RAM I could boot this kernel with: 4M (3M was too low). Not worse than 10 years back!



State of the kernel tinification project

- ▶ Stalled since Josh Triplett's patches were removed from the *linux-next* tree
- ▶ See <https://lwn.net/Articles/679455>
- ▶ Patches still available on
<https://git.kernel.org/cgit/linux/kernel/git/josh/linux.git/>
- ▶ Removing functionality through configuration settings may no longer be the way to go, as the complexity of kernel configuration parameter is already difficult to manage.
- ▶ The future may be in automatic removal of unused features (system calls, command line options, `/proc` contents, kernel command line parameters...)
- ▶ Lack of volunteers with time to drive the mainlining effort anyway.

Follow the kernel developers discussion about this topic:

<https://lwn.net/Articles/608945/>. That was in 2014!



Patches proposed by Andi Kleen in 2012

- ▶ Such optimizations would allow performance improvements as well as some size reduction by eliminating unused code (-6% on ARM, reported by Tim Bird).
- ▶ The last time the LTO patches were proposed, using LTO could create new issues or make problems harder to investigate. Linus didn't trust the toolchains at that time.
- ▶ See <https://lwn.net/Articles/512548/>



XIP: eXecution In Place

- ▶ Allows to keep the kernel text in flash (NOR flash required).
- ▶ Only workable solution for systems with very little RAM
- ▶ ARM is apparently the only platform supporting it



How to help with kernel tinification (1)

- ▶ Look for `obj-y` in kernel Makefiles:

```
obj-y      = fork.o exec_domain.o panic.o \  
            cpu.o exit.o softirq.o resource.o \  
            sysctl.o sysctl_binary.o capability.o ptrace.o user.o \  
            signal.o sys.o kmod.o workqueue.o pid.o task_work.o \  
            extable.o params.o \  
            kthread.o sys_ni.o nsproxy.o \  
            notifier.o ksysfs.o cred.o reboot.o \  
            async.o range.o smpboot.o ucount.o
```

- ▶ What about allowing to compile Linux without ptrace support (14K on arm) or without reboot (9K)?
- ▶ Another way is to look at the compile logs and check whether/why everything is needed.



How to help with kernel tinification (2)

- ▶ Look for tinification opportunities, looking for the biggest symbols:

```
nm --size-sort vmlinux
```

- ▶ Look for size regressions with the *Bloat-O-Meter*:

```
> ./scripts/bloat-o-meter vmlinux-4.9 vmlinux-4.10
```

```
add/remove: 101/135 grow/shrink: 155/109 up/down: 19517/-19324 (193)
```

function	old	new	delta
page_wait_table	-	2048	+2048
sys_call_table	-	1600	+1600
cpuhp_bp_states	980	1800	+820
...			



LLVM Linux project

<http://llvm.linuxfoundation.org/>

- ▶ Using Clang to compile the Linux kernel also opens the door to performance and size optimizations, possibly even better than what you can get with gcc LTO.
- ▶ Unfortunately, the project looks stalled since 2015.
- ▶ News: Bernhard Rosenkränzer from Linaro has updated the patchset and should start pushing upstream soon.
Reference: <https://android-git.linaro.org/kernel/hikey-clang.git>, branch `android-hikey-linaro-4.9-clang`





Userspace - BusyBox vs Toybox

```
[ 4096] bin
[ 100324] busybox
[ 7] cat -> busybox
[ 7] dmesg -> busybox
[ 7] hush -> busybox
[ 7] ls -> busybox
[ 7] mkdir -> busybox
[ 7] mount -> busybox
[ 7] mv -> busybox
[ 7] ps -> busybox
[ 7] rm -> busybox
[ 7] sh -> busybox
[ 7] umount -> busybox
[ 4096] sbin
[ 14] ifconfig -> ../bin/busybox
[ 14] init -> ../bin/busybox
[ 14] modprobe -> ../bin/busybox
[ 14] rmmod -> ../bin/busybox
2 directories, 16 files
```

```
[ 4096] bin
[ 6] cat -> toybox
[ 6] dmesg -> toybox
[ 6] ls -> toybox
[ 6] mkdir -> toybox
[ 6] mount -> toybox
[ 6] mv -> toybox
[ 6] rm -> toybox
[ 6] sh -> toybox
[ 84396] toybox
[ 6] toysh -> toybox
[ 6] umount -> toybox
[ 4096] sbin
[ 13] ifconfig -> ../bin/toybox
[ 13] init -> ../bin/toybox
[ 13] modprobe -> ../bin/toybox
[ 13] rmmod -> ../bin/toybox
[ 4096] usr
[ 4096] bin
[ 16] ps -> ../../bin/toybox
4 directories, 16 files
```

Compiled on ARM with gcc 5.4 (dynamically linked with glibc)



BusyBox vs Toybox - shell only

```
i_ [          4096] bin
  | [          59152] busybox
  | [              7] hush -> busybox
  | [              7] sh -> busybox

1 directory, 3 files
```

```
i_ [          4096] bin
  | [              6] dmesg -> toybox
  | [              6] sh -> toybox
  | [         18224] toybox
  | [              6] toysh -> toybox

1 directory, 4 files
```

Compiled on ARM with gcc 5.4 (dynamically linked with glibc)



BusyBox vs Toybox - Conclusions

- ▶ Toybox wins if your goal is to reduce size and have a tiny rootfs
- ▶ BusyBox wins in terms of configurability, and in terms of functionality for more elaborate needs.
- ▶ Comments from Rob Landley: the Toybox shell is too experimental to be used at the moment, and is meant to become a bash replacement. If you're looking for a small shell, you may look at *mksh* (<https://www.mirbsd.org/mksh.htm>)



glibc vs uclibc vs musl (static)

Let's compile and strip BusyBox 1.26.2 **statically** and compare the size

- ▶ With gcc 6.3, armel, musl 1.1.16:
183348 bytes
- ▶ With gcc 6.3, armel, uclibc-ng 1.0.22 :
210620 bytes.
- ▶ With gcc 6.2, armel, glibc:
755088 bytes

Note: BusyBox is automatically compiled with `-Os` and stripped.



glibc vs uclibc vs musl (dynamic)

Let's compile and strip BusyBox 1.26.2 **dynamically** and compare the size

- ▶ With gcc 6.3, armel, musl 1.1.16:
92948 bytes
- ▶ With gcc 6.3, armel, uclibc-ng 1.0.22 :
92116 bytes.
- ▶ With gcc 6.2, armel, glibc:
100336 bytes



glibc vs uclibc vs musl - small static executables

Let's compile and strip a `hello.c` program **statically** and compare the size

- ▶ With gcc 6.3, armel, musl 1.1.16:
7300 bytes
- ▶ With gcc 6.3, armel, uclibc-ng 1.0.22 :
67204 bytes.
- ▶ With gcc 6.2, armel, glibc:
492792 bytes



Using super strip

`sstrip` (<http://www.muppetlabs.com/~breadbox/software/elfkickers.html>) removes ELF contents that are not needed for program execution.

- ▶ Expect to save only a few hundreds or thousands of bytes
- ▶ `sstrip` is architecture independent (unlike `strip`) and is trivial to compile

Example with the small static program we've just compiled:

- ▶ With gcc 6.3, armel, musl 1.1.16: 7300 to 6520 bytes (-780)
- ▶ With gcc 6.3, armel, uclibc-ng 1.0.22: 67204 bytes to 66144 bytes (-1060)
- ▶ With gcc 6.2, armel, glibc: 492792 to 491208 bytes (-1584)

With BusyBox statically compiled with the musl library:

- ▶ From 183012 to 182289 (-723)



Other lightweight libraries

- ▶ **diet libc** (<http://www.fefe.de/dietlibc/>)
 - ▶ Latest release in 2013! Not supported by toolchain generators.
 - ▶ Was meant to generate small static executables
- ▶ **klibc** (<https://www.kernel.org/pub/linux/libs/klibc/>)
 - ▶ Latest release in 2014! Not supported by toolchain generators.
 - ▶ Was meant to generate small static executables for use in initramfs filesystems.
 - ▶ Need reviving?



Optimizing libraries

- ▶ You can use *mklibs* ([git://anonscm.debian.org/d-i/mklibs](https://anonscm.debian.org/d-i/mklibs), but that just copies the libraries which are used for a given set of executables. Build systems can already do that.
- ▶ Would need something that removes unused symbols from libraries. Is the Library Optimizer from MontaVista (<https://sourceforge.net/projects/libraryopt/>) still usable?



Achieving small filesystem size

- ▶ For very small systems, booting on an initramfs is the best solution. It allows to boot earlier and faster too (no need for filesystem and storage drivers).
- ▶ A single static executable helps too (no libraries)
- ▶ For bigger sizes, compressing filesystems are useful:
 - ▶ SquashFS for block storage
 - ▶ JFFS2 for flash (UBI has too much overhead for small partitions)
 - ▶ ZRAM (compressed block device in RAM)



Conclusions

- ▶ Though there apparently hasn't been recent mainlining efforts, the kernel size can remain very small (405K compressed on ARM, running on a system with 4M of RAM).
- ▶ Compilers: use *clang* or *gcc LTO* (not for the kernel yet)
- ▶ New C library worth using: *musl*
- ▶ Worth giving Toybox a try too, when simple command line utilities are sufficient.
- ▶ Still significant room for improvement. Difficult to make things removable without increasing the kernel parameter and testing complexity, though.



- ▶ Any recent achievements to report?
- ▶ Any other resources you are using?
- ▶ Volunteers to join the size effort?
- ▶ News from the LLVM Linux project?
- ▶ Community friendly hardware we could use for development efforts? Supporting special hardware with tight requirements is a good reason for getting code accepted.



Useful resources

- ▶ Home of the Linux tinification project <https://tiny.wiki.kernel.org/>
- ▶ Ideas ideas and projects which would be worth reviving
http://elinux.org/Kernel_Size_Reduction_Work
- ▶ Tim Bird - Advanced size optimization of the Linux kernel (2013) http://events.linuxfoundation.org/sites/events/files/lcjp13_bird.pdf
- ▶ Pieter Smith - Linux in a Lightbulb: How Far Are We on Tinification (2015)
http://www.elinux.org/images/6/67/Linux_In_a_Lightbulb-Where_are_we_on_tinification-ELCE2015.pdf
- ▶ Vitaly Wool - Linux for Microcontrollers: From Marginal to Mainstream (2015)
http://www.elinux.org/images/9/90/Linux_for_Microcontrollers-From_Marginal_to_Mainstream.pdf



Interesting talks at ELC

- ▶ Tuesday - 4:20pm
Tutorial: building the Simplest Possible Linux System - Rob Landley
- ▶ Tuesday - 5:20pm
Optimizing C for Microcontrollers - Best Practices - Khem Raj
- ▶ Thursday - 3:30pm
GCC/Clang Optimizations for Embedded Linux - Khem Raj

Questions? Suggestions? Comments?

Michael Opdenacker
michael.opdenacker@bootlin.com

Slides under CC-BY-SA 3.0

<http://bootlin.com/pub/conferences/2017/elc/opdenacker-embedded-linux-size-reduction/>



Notes from discussions with the audience (1)

- ▶ Bernhard Rosenkränzer suggested to try the Bionic C library from Android in standard Linux. It's not perfect but could be useful in some cases.
- ▶ Clang has a new `-Oz` optimization option that goes further than `-Os`
- ▶ Rob Landley mentioned his 2013 patchset to address limitations in the initramfs booting approach. See <https://lkml.org/lkml/2013/7/9/501>



Notes from discussions with the audience (2)

- ▶ In the search for a small community friendly board with very little RAM (no more than 2-4 MB of RAM), it seems that the most popular architecture is STM32.
- ▶ *Musl* library:
 - ▶ To build a *Musl* toolchain, in addition to *Crosstool-ng*, it is also possible to use the *musl-cross-make* project (<https://github.com/richfelker/musl-cross-make>)
 - ▶ *Musl* is used in the *Alpine Linux* distribution (<https://www.alpinelinux.org/>), focusing on small size and security. You could use it if your system needs a distribution.