Reducing the ARM Linux kernel size without losing your mind

Nicolas Pitre

<nicop@linaro.org>

2015-09-23

Reducing the ARM Linux kernel size

- Why?
- How?
- Problems
- Solutions
Linux kernel size reduction

Traditional Embedded Distros:

- OpenWRT
- Yocto
- etc.

Benefits:

- Longer product life
  - Software maintenance
  - New features with same hardware

Linux Kernel Size Reduction

"Internet of Things" (IoT)

Very pervasive:

- Cable/ADSL Modems
- Smart Phones
■ Smart Watches
■ Internet Connected Refrigerators
■ WI-FI-Enabled Washing Machines
■ Smart TV Sets
■ Wi-Fi enabled Light Bulbs
■ Connected Cars
■ Alarm Systems monitored via Internet
■ etc.

Linux Kernel Size Reduction

"Internet of Things" [IoT]

The problem: Security

■ All solutions will eventually be broken
  ● Think NSA...
  ● Then professional thieves...
  ● Then script kiddies...

Security Response is a must... even for gadgets!
The Internet of Things Is Wildly Insecure And Often Unpatchable


— Bruce Schneier

Linux Kernel Size Reduction

"Internet of Things" [IoT]

More problems:

- Legacy Software Maintenance is
  - Hard
  - Expensive
  - Uninteresting

Solution:

- Avoid custom software
- Leverage the Open Source community
- Gather critical mass around common infrastructure
Linux Kernel Size Reduction

Common Software Infrastructure

Linux is a logical choice

- Large community of developers
- Best looked-after network stack
- Extensive storage options
- Already widely used in embedded setups

Linux Kernel Size Reduction

Common Software Infrastructure

The Linux kernel is a logical choice... BUT

- it is featureful → Bloat
- its default tuning is for high-end systems
- the emphasis is on scaling up more than scaling down
- its flexible configuration system leads to
 Kernel Size Reduction is part of the solution

Linux Kernel Size Reduction

What can be done?

From [http://elinux.org/Kernel_Size_Reduction_Work](http://elinux.org/Kernel_Size_Reduction_Work)

- Kernel XIP
- User Space IP stack
- Kconfig options for various feature removal
- LTO build

Linux Kernel Size Reduction
LTO is cool!

- C files are parsed only
- Object files store gcc intermediate code representation
- All intermediate representations concatenated together at link time
- Optimization happens on the whole program at once

This is very effective at optimizing out unused code.

Linux Kernel Size Reduction

LTO is cool... BUT

- is still experimental
- is slow
- requires kernel changes
- is slow
- makes code debugging harder
- requires a lot of memory for successful compilation

Linux Kernel Size Reduction
LTO is cool... BUT

Table 1. Full Kernel Build Timing

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Wall Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Build</td>
<td>4m53s</td>
</tr>
<tr>
<td>LTO Build</td>
<td>10m23s</td>
</tr>
</tbody>
</table>

Table 2. Kernel Rebuild Timing After a Single Change

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Wall Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Build</td>
<td>0m12s</td>
</tr>
<tr>
<td>LTO Build</td>
<td>6m27s</td>
</tr>
</tbody>
</table>

Note

Build Details:

- Linux v4.2
- ARM multi_v7_defconfig
- gcc v5.1.0
- Intel Core2 Q6600 CPU at 2.40GHz
Linux Kernel Size Reduction

A poor man's LTO: ld -gc-sections

How it works?

Let's consider the following code:

```c
int foo(void) { return 0; }

int bar(void) { return foo(); }
```

Result:

```
.text
.type foo, %function
```


```assembly
foo:
    mov    r0, #0
    bx     lr

.type  bar, %function

bar:
    b      foo
```

Linux Kernel Size Reduction

A poor man's LTO: ld -gc-sections

How it works?

First, gcc -ffunction-sections gives separate sections to each function:

```c
int foo(void) { return 0; }  /* uses section .text.foo */

int bar(void) { return foo(); }  /* uses section .text.bar */
```
Result:

```assembly
.section .text.foo,"ax",%progbits
.type foo, %function

foo:
    .section .text.bar,"ax",%progbits
    .type bar, %function

bar:
    b foo
```

Linux Kernel Size Reduction

A poor man's LTO: ld -gc-sections

How it works?

Let's add -fdata-sections to cover global data:
int baz = 1; /* uses section .data.baz */

int foo(void) { return baz; } /* uses section .text.foo */

int bar(void) { return foo(); } /* uses section .text.bar */

Result:

```assembly
.section .text.foo,"ax",%progbits
.type foo, %function

foo:
   movw r3, #:lower16:.LANCHOR0
   movt r3, #:upper16:.LANCHOR0
   ldr r0, [r3]
   bx  lr

.section .text.bar,"ax",%progbits
.type bar, %function

bar:
   b foo

.section .data.baz,"aw",%progbits
```
Linux Kernel Size Reduction

A poor man’s LTO: ld -gc-sections

How it works?

Let’s make it into a test program:

```
#include <stdio.h>

int baz = 1;

int foo(void) { return baz; }

int bar(void) { return foo(); }

void main(void) { printf("value = %d\n", foo()); }
```
Result:

```bash
$ gcc -ffunction-sections -fdata-sections -o test test.c
$ ./test
value = 1
$ nm test | grep "foo\|bar\|baz"
00008520 T bar
00010720 D baz
000084fc T foo
```

**Linux Kernel Size Reduction**

A poor man's LTO: ld -gc-sections

**How it works?**

Finally, pass -gc-sections to the linker:
```bash
$ gcc -ffunction-sections -fdata-sections \ 
  -Wl,-gc-sections -Wl,-print-gc-sections \ 
  -o test test.c
ld: Removing unused section '.text.bar' in file 'test.o'
$ nm test | grep "foo\|bar\|baz"
  000106fc D baz
  000084fc T foo
```

**Linux Kernel Size Reduction**

A poor man's LTO: ld -gc-sections

What about the Linux kernel?

Let's start with the smallest kernel:

```bash
$ make allnoconfig
```
$ make vmlinux
$ size vmlinux

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>774608</td>
<td>71024</td>
<td>14876</td>
<td>860508</td>
<td>d215c vmlinux</td>
</tr>
</tbody>
</table>

What else can we configure out?

**Linux Kernel Size Reduction**

A poor man’s LTO: `ld -gc-sections`

Let’s also disable all system calls:

```diff
diff --git a/include/linux/syscalls.h b/include/linux/syscalls.h
index b45c45b8c8..8868b7b7b7 100644
--- a/include/linux/syscalls.h
+++ b/include/linux/syscalls.h
@@ -198,7 +198,10 @@ extern struct
trace_event_functions exit syscall_print_funcs;
 asmlinkage long
 SyS##name(__MAP(x,__SC_LONG,__VA_ARGS__));
```
asmlinkage long
SyS##name(__MAP(x,__SC_LONG,__VA_ARGS__))  \\
{
  \\
  -  long ret =
SyS##name(__MAP(x,__SC_CAST,__VA_ARGS__));  \\
+  long ret;
  \\
+  if
(IS_ENABLED(CONFIG_NO_SYSCALLS))  \\
  +  return -ENOSYS;
  \\
  +  ret =
SyS##name(__MAP(x,__SC_CAST,__VA_ARGS__));  \\
    __MAP(x,__SC_TEST,__VA_ARGS__);  \\
    __PROTECT(x,
ret,__MAP(x,__SC_ARGS,__VA_ARGS__));  \\
    return ret;
  \\

Linux Kernel Size Reduction

A poor man’s LTO: ld -gc-sections
Let's also disable all system calls:

Table 3. Size of the vmlinux binary

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Size [bytes]</th>
<th>Reference %</th>
</tr>
</thead>
<tbody>
<tr>
<td>allnoconfig</td>
<td>860508</td>
<td>100%</td>
</tr>
<tr>
<td>allnoconfig + CONFIG_NO_SYSCALLS</td>
<td>815804</td>
<td>94.8%</td>
</tr>
</tbody>
</table>

Still **way** too big for a kernel that can't do anything.

**Linux Kernel Size Reduction**

A poor man's LTO: `ld -gc-sections`

Let's apply `-gc-sections` to the kernel:

diff --git a/Makefile b/Makefile
index 342678e36c..80b86d72d4 100644
--- a/Makefile
+++ b/Makefile
@@ -630,6 +630,10 @@ else
     KBUILD_CFLAGS += -02
 endif

+ifdef CONFIG_GC_SECTIONS
+KBUILD_CFLAGS += -ffunction-sections
-fdata-sections
+endif
+
+ # Tell gcc to never replace conditional load
with a non-conditional one
     KBUILD_CFLAGS += $(call cc-option,--param=allow-store-data-races=0)

@@ -820,6 +824,10 @@ ifeq
     ($(CONFIG_STRIP_ASM_SYMS),y)
     LDFLAGS_vmlinux += $(call ld-option, -X,)
 endif

+ifdef CONFIG_GC_SECTIONS
+LDFLAGS_vmlinux += -gc-sections
+endif
+
+ # Default kernel image to build when no specific
target is given.
+ # KBUILD_IMAGE may be overruled on the command
line or
+ # set in the environment
Linux Kernel Size Reduction

A poor man’s LTO: ld -gc-sections

Can’t be hard, right?

$ make vmlinux
[...]
  CC     init/version.o
  LD     init/built-in.o
  LD     vmlinux

arm-linux-ld: missing CPU support
arm-linux-ld: no machine record defined
Makefile:963: recipe for target 'vmlinux' failed

Does this ring a bell?
A poor man's LTO: ld -gc-sections

From arch/arm/kernel/vmlinux.lds:

/*
 * These must never be empty
 * If you have to comment these two assert statements out, your
 * binutils is too old (for other reasons as well)
 */

ASSERT((__proc_info_end - __proc_info_begin),
"missing CPU support")

ASSERT((__arch_info_end - __arch_info_begin), "no machine record defined")

They come from:

.init.proc.info : {
    . = ALIGN(4);
Linux Kernel Size Reduction

A poor man's LTO: ld -gc-sections

From arch/arm/include/asm/mach/arch.h:

/*
   * Set of macros to define architecture features.
   * This is built into
   * a table by the linker.
   */
#define MACHINE_START(_type,_name)
static const struct machine_desc
__mach_desc_##_type \       __used
/  
__attribute__((__section__(".arch.info.init"))) = 
{ \  
    .nr = MACH_TYPE_##_type, \  
    .name = _name, \}

#define DT_MACHINE_START(_name, _namestr) \  
static const struct machine_desc
__mach_desc_##_name \      __used
/  
__attribute__((__section__(".arch.info.init"))) = 
{ \  
    .nr = ~0, \  
    .name = _namestr, \   

#define MACHINE_END \  }
  

Linux Kernel Size Reduction
A poor man's LTO: ld -gc-sections

Example usage:

```c
MACHINE_START(EBSA110, "EBSA110")
/* Maintainer: Russell King */
.atan_offset = 0x400,
.reserve_lpo = 1,
.reserve_lpo2 = 1,
.map_io = ebsa110_map_io,
.init_early = ebsa110_init_early,
.init_irq = ebsa110_init_irq,
.init_time = ebsa110_timer_init,
.restart = ebsa110_restart,
MACHINE_END
```

Linux Kernel Size Reduction

We have to prevent those table sections with no explicit references from
being garbage collected:

```c
diff --git a/arch/arm/kernel/vmlinux.lds.S b/arch/arm/kernel/vmlinux.lds.S
index 8b60fde5ce..d62ccc2972 100644
--- a/arch/arm/kernel/vmlinux.lds.S
+++ b/arch/arm/kernel/vmlinux.lds.S
@@ -15,7 +15,7 @@
     #define PROC_INFO

     . = ALIGN(4);

-VMLINUX_SYMBOL(__proc_info_begin) = .;
-VMLINUX_SYMBOL(__proc_info_end) = .;

#define IDMAP_TEXT
@@ -187,7 +187,7 @@ SECTIONS
}
    .init.arch.info : {
        __arch_info_begin = .;
-       *(.arch.info.init)
+       KEEP(*(.arch.info.init))
        __arch_info_end = .;
```
Can't be **that** hard, right?

**Linux Kernel Size Reduction**

---

A poor man's LTO: `ld -gc-sections`

---

Can't be **that** hard, right?

```bash
$ make vmlinux

[...]
CC       init/version.o
LD       init/built-in.o
LD       vmlinux
SYSMAP   System.map
$

Success!
```
Linux Kernel Size Reduction

A poor man’s LTO: ld -gc-sections

How effective on the kernel?

Table 4. Size of the vmlinux binary

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Size [bytes]</th>
<th>Reference %</th>
</tr>
</thead>
<tbody>
<tr>
<td>allnoconfig</td>
<td>860508</td>
<td>100%</td>
</tr>
<tr>
<td>allnoconfig + CONFIG_NO_SYSCALLS</td>
<td>815804</td>
<td>94.8%</td>
</tr>
<tr>
<td>allnoconfig + CONFIG_NO_SYSCALLS +</td>
<td>555798</td>
<td>64.6%</td>
</tr>
<tr>
<td>CONFIG_GC_SECTIONS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What about authentic LTO?
Linux Kernel Size Reduction

A poor man’s LTO: ld -gc-sections

Let’s compare against authentic LTO:

Table 5. Size of the vmlinux binary

<table>
<thead>
<tr>
<th>Build Type</th>
<th>Size [bytes]</th>
<th>Reference %</th>
</tr>
</thead>
<tbody>
<tr>
<td>allnoconfig</td>
<td>860508</td>
<td>100%</td>
</tr>
<tr>
<td>allnoconfig + CONFIG_NO_SYSCALLS</td>
<td>815804</td>
<td>94.8%</td>
</tr>
<tr>
<td>allnoconfig + CONFIG_NO_SYSCALLS + CONFIG_GC_SECTIONS</td>
<td>555798</td>
<td>64.6%</td>
</tr>
<tr>
<td>allnoconfig + CONFIG_NO_SYSCALLS + CONFIG_LTO</td>
<td>488264</td>
<td>56.7%</td>
</tr>
</tbody>
</table>

The -gc-sections result is somewhat bigger but so much faster.
Still... it’s big for a kernel that does nothing...

**Linux Kernel Size Reduction**

---

A poor man’s LTO: `ld -gc-sections`

---

Is that it?

**Linux Kernel Size Reduction**

---

A poor man’s LTO: `ld -gc-sections`

---

More table sections have to be marked with `KEEP()`:

- the initcall pointer table
- the exception fixup pointer table
- the vector table and stubs
- the pa/va code patching pointer table
- the SMP alt code pointer table
Linux Kernel Size Reduction

A poor man's LTO: ld -gc-sections

The backward reference problem:

```c
int foobar(int __user *p)
{
    return put_user(0x5a, p);
}
```

Result:

```
1:    .section .text.foobar,"ax"
2:    foobar:
```
Linux Kernel Size Reduction

A poor man's LTO: ld -gc-sections

The "backward reference" problem:

1: .section .text.fool,"ax"
2: fool: ...
3: 
4: .section .text.fool2,"ax"
Linux Kernel Size Reduction

A poor man's LTO: ld -gc-sections

A closer look at put_user():
A poor man’s LTO: `ld -gc-sections`

A closer look at `put_user()`:
#define __put_user_err(x, ptr, err)

do {
    unsigned long __pu_addr = (unsigned long)(ptr);
    __typeof__(*(ptr)) __pu_val = (x);
    __chk_user_ptr(ptr);
    might_fault();
    switch (sizeof(*ptr)) {
        case 1: __put_user_asm_byte(__pu_val, __pu_addr, err); break;
        case 2: __put_user_asm_half(__pu_val, __pu_addr, err); break;
        case 4: __put_user_asm_word(__pu_val, __pu_addr, err); break;
        case 8: __put_user_asm_dword(__pu_val, __pu_addr, err); break;
        default: __put_user_bad();
    }
} while (0)
Linux Kernel Size Reduction

A poor man's LTO: ld -gc-sections

A closer look at put_user():

```c
#define __put_user_asm_word(x, __pu_addr, err)

__asm__ __volatile__ (
    "1:    " TUSER(str) " \%1,[\%2],\%0\n",
    "2:\n",
    "    .pushsection .text.fixup,"ax"\n",
    "    .align 2\n",
    "3:    mov \%0, \%3\n",
    "    b 2b\n",
    "    .popsection\n"
)
```
How to create distinct .text.fixup and __ex_table section instances?

**Linux Kernel Size Reduction**

Context based ELF section creation

Some possibilities:

```c
__put_user(val, ptr, __func__)
```
__put_user(val, ptr, __FUNCTION__) 

__put_user(val, ptr, __PRETTY_FUNCTION__) 

__put_user(val, ptr, __FILE__, __LINE__) 

__put_user(val, ptr, __COUNTER__) 

Linux Kernel Size Reduction  

Context based ELF section creation 

The solution: modify gas

.macro exception_code
.pushsection %s.exception
...
.popsection
.endm
Resulting sections:

- .text.foo.exception
- .text.bar.exception

**Note**
Feature available upstream, in binutils 2.25.51.0.3 from H.J. Lu and Linaro release.

**Linux Kernel Size Reduction**

Context based ELF section creation
This fixes the built-in \_exit section problem:

```c
1:      .text
2:     foobar:
3:  1:       ...
4:
5:      \section .init.text
6:   foobar_init:
7:  2:       ...
8:
9:      \section .exit.text
10:  foobar_exit:
11:  3:       ...
12:
13:      \section .text.fixup
14:   do_fix  1b
15:
16:      \section .init.text.fixup
17:   do_fix  2b
18:
19:      \section .exit.text.fixup
20:   do_fix  3b
```

**Linux Kernel Size Reduction**
Context based ELF section creation

The "missing forward reference" problem:

```
1:  .section .text.foobar,"ax"
2:  .section .fixup.text.foobar,"ax"
3:  mov r3, #0
4:  mov r2, #0x5a
5:  1:  str r2, [r0]
6:  2:  mov r0, r3
7:  bx lr
8:  .section  __ex_table.text.foobar,"a"
9:  .long  1b, 3b
```
Linux Kernel Size Reduction

Context based ELF section creation

The "missing forward reference" solution:

```assembly
 1: .section .text.foobar,"ax"
 2: foobar:
 3:   mov    r3, #0
 4:   mov    r2, #0x5a
 5:   str    r2, [r0]
 6:   .tug   4f
 7:   mov    r0, r3
 8:   bx     lr
 9:
10: .section .fixup.text.foobar,"ax"
11: 3:    mov    r3, #-EFAULT
12:    b      2b
13:
14: .section
__ex_table.text.foobar,"a"
15: 4:    .long   1b, 3b
```
Linux Kernel Size Reduction

Context based ELF section creation

The "missing forward reference" solution:

Turns out that no modifications to gas is necessary.

```
1:      .section .text.foobar,"ax"
2:      .section .fixup.text.foobar,"ax"
3:      mov  r3, #0
4:      mov  r2, #0x5a
5:      str  r2, [r0]
6:      .reloc ., R_ARM_NONE, 4f
7:      mov  r0, r3
8:      bx   lr
9:      b     2b
10:     .section .__ex_table.text.foobar,"a"
11:     .section _ex_table.text.foobar,"a"
```
Linux Kernel Size Reduction

Making both LTO and -gc-sections effective

Reducing number of "Peg Point" Symbols:

- configurable system calls
- syscalls as modules
- selective EXPORT_SYMBOL()