Internet of Tiny Linux (IoTL): the sequel.

Nicolas Pitre
<nico@linaro.org>
2016-09-29
Nicolas Pitre

This is a discussion on various methods put forward to reduce the size of Linux kernel and user space binaries to make them suitable for small IoT applications, ranging from low hanging fruits to more daring approaches. Results from on-going work will also be presented.
"Internet of Things" (IoT)

Goals:

- Ubiquitous and Low Cost
- Reliable and Easy to Use
- Secure and Field-Upgradable

Pick two!
"Internet of Things" (IoT)

Solutions:

- Avoid custom base software
- Leverage the Open Source community
- Gather critical mass around common infrastructure
- Share the cost of non-differentiating development
Internet of Tiny Linux (IoTL)

"Internet of Things" (IoT)

**Linux** is a logical choice

- Large community of developers
- Best looked-after network stack
- Extensive storage options
- Already widely used in embedded setups
- Etc.
Internet of Tiny Linux (IoTL)

"Internet of Things" (IoT)

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
  - Kconfig hell
  - suboptimal build
- is the largest component in most Linux-based embedded systems

Linux Kernel Size Reduction is part of the solution
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

Automatic size reduction techniques:

- Linker Section Garbage Collection (-gc-sections)
- Link Time Optimization (LTO)
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.
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

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

```c
int foo(void) { return 123; }
/* uses section .text.foo */

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

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

Result:

```assembly
.section .text.foo,"ax",%progbits
.type foo, %function
```
foo:
  mov   r0, #123
  bx    lr

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

bar:
  bl     foo
  bx     lr

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

main:
  bl     foo
  bx     lr
Linux Kernel Size Reduction

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

How it works?

Then, pass `-gc-sections` to the linker:

```bash
$ gcc -ffunction-sections \\
>   -Wl,-gc-sections \\
>   -Wl,-print-gc-sections \\
>   -o test test.c
ld: Removing unused section '.text.bar' in file '/tmp/cc2QRsIs.o'

$ nm test | grep "foo\|bar\|main"
000103a1 T foo
000103b1 T main
```
Internet of Tiny Linux (IoTL)

A poor man’s LTO: `ld -gc-sections`

Kernel image size comparison

### 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>
<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 to build.
Reducing the Linux Kernel Size

Still…

- Why up to 480KB of kernel code remains when everything is configured out?
- What is taking so much space?
- Why isn’t LTO or -gc-sections dropping more code?
Reducing the Linux Kernel Size

So many anchors, so many ropes

- __initcall() and variants:
  - early_initcall()
  - pure_initcall()
  - core_initcall()
  - postcore_initcall()
  - arch_initcall()
  - subsys_initcall()
  - fs_initcall()
  - rootfs_initcall()
  - device_initcall()
  - late_initcall()

- More initcall variants:
  - console_initcall()
- security_initcall()
- module_init()
Reducing the Linux Kernel Size

So many anchors, so many ropes

The actual count:

```
$ make tinyconfig && make
$ awk --non-decimal-data '>
> / __initcall_start$/ { s = ("0x" $1)+0 }
> / __initcall_end$/    { e = ("0x" $1)+0 }
> END { print (e - s)/4 }' System.map
49
```
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

More anchors, more ropes

- __setup_param()
- early_param()
- module_param()

Counts:

```bash
$ awk --non-decimal-data '>
> / __setup_start$/ { s = ("0x" $1)+0 }
> / __setup_end$/ { e = ("0x" $1)+0 }
> END { print (e - s)/12 } ' System.map
47
```

```bash
$ awk --non-decimal-data '>
> / __start___param$/ { s = ("0x" $1)+0 }
> / __stop___param$/ { e = ("0x" $1)+0 }
```
> END { print (e - s)/20 } ' System.map
12
Reducing the Linux Kernel Size

The EXPORT_SYMBOL anchor problem

- EXPORT_SYMBOL(foo_bar) forces foo_bar() into the kernel even if there is no users.
- Let’s export only those symbols needed by the set of configured modules.

Available in Linux v4.7: "Trim unused exported kernel symbols"

CONFIG_TRIM UNUSED_KSYMS

The kernel and some modules make many symbols available for other modules to use via EXPORT_SYMBOL() and variants. Depending on the set of modules being selected in your kernel configuration, many of those exported symbols might never be used.

This option allows for unused exported symbols to be dropped from the build. In turn, this provides the compiler more opportunities (especially when using LTO) for optimizing the code and reducing binary size. This might have some security advantages as well.
Reducing the Linux Kernel Size

Trim unused exported kernel symbols

Size numbers from a smallish configuration:

Kernel v4.5-rc2 ARM realview_defconfig + LTO

```
$ size vmlinux
   text     data      bss    dec    hex   filename
 4422942   209640  126880  4759462  489fa6    vmlinux

$ wc -l Module.symvers
  5597 Module.symvers
```

Kernel v4.5-rc2 ARM realview_defconfig + LTO + CONFIG_TRIM_UNUSED_KSYMS

```
$ size vmlinux
   text     data      bss    dec    hex   filename
 3823485   205416  125800  4154701  3f654d    vmlinux
```
Here we reduced the kernel text by about 13.6%. Disabling module support altogether would reduce it by 13.9% i.e. only 0.3% difference.

This means the overhead of using modules on embedded targets is greatly reduced.
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

Subsystem specific trimming

Already Done (few examples):

- Compile out block device support.
- Compile out NTP support.
- Compile out support for capabilities (only root is granted permission).
- Compile out support for non-root users and groups.
- Compile out printk() and related strings.
- Etc.

Needed: more similar config options.
# Internet of Tiny Linux (IoTL)

## Reducing the Linux Kernel Size

### What to look for?

```bash
$ make clean && make tinyconfig && make
$ size $(find . -name built-in.o) | sort -n
```

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>196</td>
<td>0</td>
<td>256</td>
<td>100</td>
<td>./drivers/clocks/source/built-in.o</td>
</tr>
<tr>
<td>212</td>
<td>0</td>
<td>0</td>
<td>212</td>
<td>d4</td>
<td>./drivers/base/power/built-in.o</td>
</tr>
<tr>
<td>641</td>
<td>196</td>
<td>4</td>
<td>841</td>
<td>349</td>
<td>./drivers/irqchip/built-in.o</td>
</tr>
<tr>
<td>716</td>
<td>0</td>
<td>0</td>
<td>716</td>
<td>2cc</td>
<td>./drivers/rtc/built-in.o</td>
</tr>
<tr>
<td>1508</td>
<td>372</td>
<td>48</td>
<td>1928</td>
<td>788</td>
<td>./kernel/power/built-in.o</td>
</tr>
<tr>
<td>2128</td>
<td>0</td>
<td>0</td>
<td>2128</td>
<td>850</td>
<td>./security/built-in.o</td>
</tr>
<tr>
<td>2140</td>
<td>32</td>
<td>4</td>
<td>2176</td>
<td>880</td>
<td>./fs/ramfs/built-in.o</td>
</tr>
<tr>
<td>2182</td>
<td>112</td>
<td>248</td>
<td>2542</td>
<td>9ee</td>
<td>./kernel/printk/built-in.o</td>
</tr>
<tr>
<td>2296</td>
<td>24</td>
<td>0</td>
<td>2320</td>
<td>910</td>
<td>./kernel/rcu/built-in.o</td>
</tr>
<tr>
<td>4331</td>
<td>0</td>
<td>0</td>
<td>4331</td>
<td>10eb</td>
<td>./kernel/locking/built-in.o</td>
</tr>
<tr>
<td>4579</td>
<td>1233</td>
<td>12</td>
<td>5824</td>
<td>16c0</td>
<td>./arch/arm/mm/built-in.o</td>
</tr>
<tr>
<td>5117</td>
<td>14000</td>
<td>64</td>
<td>19181</td>
<td>4aed</td>
<td>./init/built-in.o</td>
</tr>
<tr>
<td>6374</td>
<td>324</td>
<td>844</td>
<td>7542</td>
<td>1d76</td>
<td>./drivers/char/built-in.o</td>
</tr>
<tr>
<td>14683</td>
<td>3148</td>
<td>224</td>
<td>18055</td>
<td>4687</td>
<td>./arch/arm/kernel/built-in.o</td>
</tr>
<tr>
<td>15379</td>
<td>572</td>
<td>2064</td>
<td>18015</td>
<td>465f</td>
<td>./kernel/irq/built-in.o</td>
</tr>
<tr>
<td>Size</td>
<td>17960</td>
<td>336</td>
<td>484</td>
<td>18780</td>
<td>495c</td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>Size</td>
<td>19186</td>
<td>560</td>
<td>24</td>
<td>19770</td>
<td>4d3a</td>
</tr>
<tr>
<td>Size</td>
<td>20422</td>
<td>1312</td>
<td>120</td>
<td>21854</td>
<td>555e</td>
</tr>
<tr>
<td>Size</td>
<td>27439</td>
<td>1288</td>
<td>108</td>
<td>28835</td>
<td>70a3</td>
</tr>
<tr>
<td>Size</td>
<td>36516</td>
<td>2956</td>
<td>3912</td>
<td>43384</td>
<td>a978</td>
</tr>
<tr>
<td>Size</td>
<td>59621</td>
<td>76</td>
<td>49</td>
<td>59746</td>
<td>e962</td>
</tr>
<tr>
<td>Size</td>
<td>72380</td>
<td>2900</td>
<td>1492</td>
<td>76772</td>
<td>12be4</td>
</tr>
<tr>
<td>Size</td>
<td>77728</td>
<td>4232</td>
<td>3532</td>
<td>85492</td>
<td>14df4</td>
</tr>
<tr>
<td>Size</td>
<td>107929</td>
<td>824</td>
<td>5540</td>
<td>114293</td>
<td>1be75</td>
</tr>
<tr>
<td>Size</td>
<td>145821</td>
<td>10136</td>
<td>8264</td>
<td>164221</td>
<td>2817d</td>
</tr>
</tbody>
</table>

Some sizes are clearly excessive for a kernel that can’t do much.
Reducing the Linux Kernel Size

Let’s remove POSIX timers

CONFIG_POSIX_TIMERS

This includes native support for POSIX timers to the kernel. Most embedded systems may have no use for them and therefore they can be configured out to reduce the size of the kernel image.

When this option is disabled, the following syscalls won’t be available: timer_create, timer_gettime: timer_getoverrun, timer_settime, timer_delete, clock_adjtime. Furthermore, the clock_settime, clock_gettime, clock_getres and clock_nanosleep syscalls will be limited to CLOCK_REALTIME and CLOCK_MONOTONIC only.
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

Let’s remove POSIX timers

**CONFIG_POSIX_TIMERS=y**

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>36516</td>
<td>2956</td>
<td>3912</td>
<td>43384</td>
<td>a978 ./kernel/time/built-in.o</td>
</tr>
</tbody>
</table>

**CONFIG_POSIX_TIMERS=n**

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>27329</td>
<td>2884</td>
<td>1200</td>
<td>31413</td>
<td>7ab5 ./kernel/time/built-in.o</td>
</tr>
</tbody>
</table>

Standard distros happen to still boot fine with CONFIG_POSIX_TIMERS=n.
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

What else can be trimmed?

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>2182</td>
<td>112</td>
<td>248</td>
<td>2542</td>
<td>9ee ./kernel/printk/built-in.o</td>
</tr>
</tbody>
</table>

Despite CONFIG_PRINTK=n there is still 2542 bytes of memory used.
Reducing the Linux Kernel Size

What else can be trimmed?

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>4331</td>
<td>0</td>
<td>0</td>
<td>4331</td>
<td>10eb ./kernel/locking/built-in.o</td>
</tr>
</tbody>
</table>

So much locking code for a uniprocessor kernel?
Reducing the Linux Kernel Size

What else can be trimmed?

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>15379</td>
<td>572</td>
<td>2064</td>
<td>18015</td>
<td>465f ./kernel/irq/built-in.o</td>
</tr>
</tbody>
</table>

There ought to be a way to acknowledge interrupts with less code in a small and static system, right?
Reducing the Linux Kernel Size

What else can be trimmed?

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>72380</td>
<td>2900</td>
<td>1492</td>
<td>76772</td>
<td>12be4 ./drivers/built-in.o</td>
</tr>
</tbody>
</table>

Isn’t it supposed to be the tiniest config?

How can still be so much driver code left?
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

What else can be trimmed?

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>77728</td>
<td>4232</td>
<td>3532</td>
<td>85492</td>
<td>14df4 ./mm/built-in.o</td>
</tr>
</tbody>
</table>

So much memory management code for a no-MMU build?
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

What else can be trimmed?

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>107929</td>
<td>824</td>
<td>5540</td>
<td>114293</td>
<td>1be75 ./fs/built-in.o</td>
</tr>
</tbody>
</table>

And the jackpot: over 110KB of memory usage for a kernel that has not any filesystem configured in.
Reducing the Linux Kernel Size

Trimming alone won’t cut it.

Substitution of subsystems must be considered.

Alternatives already exist for the memory allocator:

- SLAB
- SLUB (Unqueued Allocator)
- SLOB (Simple Allocator)

CONFIG_SLOB

SLOB replaces the stock allocator with a drastically simpler allocator. SLOB is generally more space efficient but does not perform as well on large systems.
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

What about

…

the scheduler?
Reducing the Linux Kernel Size

Some numbers for the scheduler:

```bash
$ wc -l kernel/sched/*.[ch] | sort -n
[...] 397 kernel/sched/loadavg.c 532 kernel/sched/cpufreq_schedutil.c 624 kernel/sched/wait.c 904 kernel/sched/cputime.c 973 kernel/sched/debug.c 1813 kernel/sched/deadline.c 1824 kernel/sched/sched.h 2371 kernel/sched/rt.c 8622 kernel/sched/core.c 8789 kernel/sched/fair.c 29996 total
```

```bash
$ make defconfig && make kernel/sched/
$ size kernel/sched/built-in.o
```
<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>137899</td>
<td>3465</td>
<td>1204</td>
<td>142568</td>
<td>22ce8</td>
</tr>
</tbody>
</table>

Definitely large and complex.
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

The SLOB of schedulers: nanosched

Goals:

- Smallest code base possible
- No Frills
- Dispense with SMP support
- Simplistic scheduling policy
- Minimal user space interfaces
- Optimized for small number of tasks
- Able to run standard user space distros
- Leave the standard scheduler alone!
**Internet of Tiny Linux (IoTL)**

**Reducing the Linux Kernel Size**

The SLOB of schedulers: nanosched

Implementation:

- Based on BFS from Con Kolivas
- Axed 2/3 of the original code
- Hardcoded a single scheduling policy
- Made as unintrusive as possible
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

The SLOB of schedulers: nanosched

Numbers:

```bash
$ git diff --shortstat
  12 files changed, 3051 insertions(+), 5 deletions(-)
```

```bash
$ make tinyconfig && make kernel/sched/
$ size kernel/sched/built-in.o

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>20325</td>
<td>1308</td>
<td>120</td>
<td>21753</td>
<td>54f9</td>
<td>kernel/sched/built-in.o</td>
</tr>
</tbody>
</table>
```

```bash
$ echo CONFIG_SCHED_NANO=y >> .config
$ make kernel/sched/
$ size kernel/sched/built-in.o

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>9853</td>
<td>324</td>
<td>208</td>
<td>10385</td>
<td>2891</td>
<td>kernel/sched/built-in.o</td>
</tr>
</tbody>
</table>
```
And yes, a kernel with nanosched can boot standard distros with no apparent issues.

Bets are open for eventual inclusion into mainline!
Reducing the Linux Kernel Size

Alternative to the networking subsystem:

- Substitute the current stack for a much smaller one preserving most of the socket API.
- Might not work for Apache or Samba, but simpler client-type applications could just work with either stacks.
- No need for 10gbps packet routing, or any routing for that matter.
- Still allow for seamless code development and testing on a regular Linux workstation while being 100x smaller.

Best solution might turn out to be a socket that only supports passthrough of raw frames to existing hardware drivers, with a minimal or custom IP stack in user space.
Reducing the Linux Kernel Size

Alternative to the VFS subsystem:

- No caching
- Minimal concurrency control for correctness
- Optimized for serial access from a single thread

Best solution might turn out to be direct I/O to block device drivers with the filesystem code in user space, à la mtools.
Internet of Tiny Linux (IoTL)

Reducing the Linux Kernel Size

Overall vision for a tinified Linux system

- The Linux environment and programming model is mastered by a LOT of people.
- Kernel subsystem alternatives could be developed, tested and validated by anyone on any hardware, greatly simplifying maintenance.
- The actual IoT applications could be developed, tested and validated by anyone on any hardware (even with memory protection and existing debugging tools), greatly lowering development costs.
- Existing security penetration tools, fuzzers, regression frameworks and similar infrastructure could be leveraged with minimal changes.
- Extensive hardware coverage, etc.

But all this needs to be merged into the mainline source tree to be sustainable.
Internet of Tiny Linux (IoTL)

Questions?