Tiny Linux Kernel Project: Section Garbage Collection Patchset

Wu Zhangjin
Tiny Lab - Embedded Geeks
http://tinylab.org
wuzhangjin@gmail.com

Sheng Yong
Distributed & Embedded System Lab, SISE, Lanzhou University, China
Tianshui South Road 222, Lanzhou, P.R.China
shengy07@lzu.edu.cn

Abstract

Linux is widely used in embedded systems which always have storage limitation and hence requires size optimization. In order to reduce the kernel size, based on the previous work of the "Section Garbage Collection Patchset", This paper focuses on details its principle, presents some new ideas, documents the porting steps, reports the testing results on the top 4 popular architectures: ARM, MIPS, PowerPC, X86 and at last proposes future works which may enhance or derive from this patchset.

1 Introduction

The size of Linux kernel source code increases rapidly, while the memory and storage are limited in embedded systems (e.g. in-vehicle driving safety systems, data acquisition equipments etc.). This requires small or even tiny kernel to lighten or even eliminate the limitation and eventually expand the potential applications.

Tiny Lab estimated the conventional tailoring methods and found that the old Tiny-linux project is far from being finished and requires more work and hence submitted a project proposal to CELF: "Work on Tiny Linux Kernel" to improve the previous work and explore more ideas.

"Section garbage collection patchset(gc-sections)" is a subproject of Tiny-linux, the initial work is from Denys Vlasenko[9]. The existing patchset did make the basic support of section garbage collection work on X86 platforms, but is still outside of the mainline for the limitation of the old GNU toolchains and for there are some works to be done(e.g. compatibility of the other kernel features).

Our gc-sections subproject focuses on analyzes its working mechanism, improves the patchset(e.g. unique user defined sections), applies the ideas for more architectures, tests them and explores potential enhancement and derivation. The following sections will present them respectively.

2 Link time dead code removing using section garbage collection

Compiler puts all executable code produced by compiling C codes into section called .text, r/o data into section called .rodata, r/w data into .data, and uninitialized data into .bss[2, 3]. Linker does not know which parts of sections are referenced and which ones are not referenced. As a result, unused(or 'dead') function or data cannot be removed. In order to solve this issue, each function or data should has its own section.

gcc provides `-ffunction-sections` or `-fdata-sections` option to put each function or data to its own section, for instance, there is a function called unused_func(), it goes to .text.unused_func section, Then, *ld* provides the `--gc-sections` option to check the references and determine which dead function or data should be removed, and the `--print-gc-sections` option of *ld* can print the the function or data being removed, which is helpful to debugging.

The following two figures demonstrates the differences between the typical object and the one with `-ffunction-sections`:

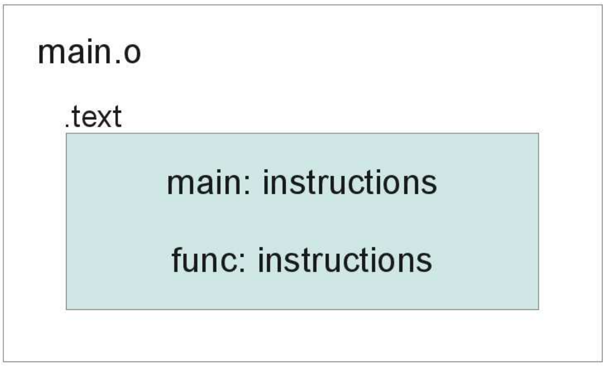

FIGURE 1: *Typical Object*

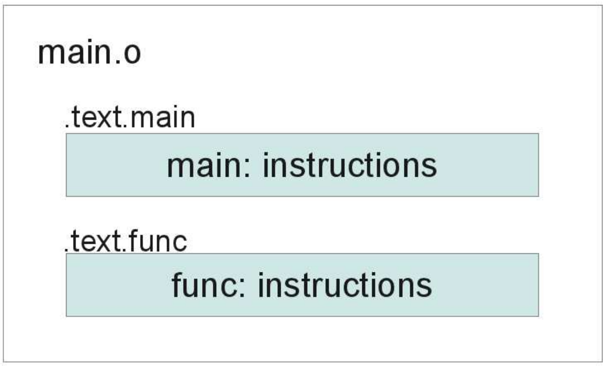

FIGURE 2: *Object with -ffunction-sections*

To learn more about the principle of section garbage collection, the basic compiling, assebmling and linking procedure should be explained at first (Since the handling of data is similar to function, will mainly present function below).

2.1 Compile: Translate source code from C to assembly

If no `-ffunction-sections` for gcc, all functions are put into .text section (indicated by the .text instruction of assembly):

```
$ echo ’unused(){} main(){}’ | gcc -S -x c -o - - \
  | grep .text
        .text
```

Or else, each function has its own section (indicated by the .section instruction of assembly):

```
$ echo ’unused(){} main(){}’ \
 | gcc -ffunction-sections -S -x c -o - - | grep .text
        .section        .text.unused,"ax",@progbits
        .section        .text.main,"ax",@progbits
```

As we can see, the prefix is the same .text, the suffix is function name, this is the default section naming rule of gcc.

Expect `-ffunction-sections`, the section attribute instruction of gcc can also indicate where should a section of the funcion or data be put in, and if it is used together with `-ffunction-sections`, it has higher priority, for example:

```
$ echo ’__attribute__ ((__section__(".text.test"))) unused(){} \
  main(){}’ | gcc -ffunction-sections -S -x c -o - - | grep .text
        .section        .text.test,"ax",@progbits
        .section        .text.main,"ax",@progbits
```

.text.test is indicated instead of the default .text.unused. In order to avoid function redefinition, the function name in a source code file should be unique, and if only with `-ffunction-sections`, every function has its own unique section, but if the same section attribute applies on different functions, different functions may share the same section:

```
$ echo ’__attribute__ ((__section__(".text.test"))) unused(){} \
  __attribute__ ((__section__(".text.test"))) main(){}’ \
  | gcc -ffunction-sections -S -x c -o - - | grep .text
        .section        .text.test,"ax",@progbits
```

Only one section is reserved, this breaks the core rule of section garbage collection: *before linking, each function or data should has its own section*, but sometimes, for example, if want to call some functions at the same time, section attribute instruction is required to put these functions to the same section and call them one by one, but how to meet these two requirements? use the section attribute instruction to put the functions to the section named with the same prefix but unique suffix, and at the linking stage, merge the section which has the same prefix to the same section, so, to linker, the sections are unique and hence better for dead code elimination, but still be able to link the functions to the same section. The implementation will be explained in the coming sections.

Based on the same rule, the usage of section attribute instruction should also follow the other two rules:

1. The section for function should named with .text prefix, then, the linker may be able to merge all of the .text sections. or else, will not

be able to or not conveniently merge the sections and at last instead may increase the size of executable for the accumulation of the section alignment.

2. The section name for function should be prefixed with .text. instead of the default .text prefix used gcc and break the core rule.

And we must notice that: ‘You will not be able to use “gprof” on all systems if you specify this option and you may have problems with debugging if you specify both this option and -g.’ (gcc man page)

```
$ echo ’unused(){} main(){}’ | \
  gcc -ffunction-sections -p -x c -o test -
<stdin>:1:0: warning: -ffunction-sections disabled; \
        it makes profiling impossible
```

2.2 Assemble: Translate assembly files to binary objects

In assembly file, it is still be possible to put the function or data to an indicated section with the .section instruction (.text equals .section “.text”). Since `-ffunction-sections` and `-fdata-sections` doesn’t work for assembly files and they has no way to determine the function or data items, therefore, for the assembly files written from scratch (not translated from C language), .section instruction is required to added before the function or data item manually, or else the function or data will be put into the same .text or .data section and the section name indicated should also be unique to follow the core rule of section garbage collection.

The following commands change the section name of the ‘unused’ function in the assembly file and show that it does work.

```
$ echo ’unused(){} main(){}’ \
  | gcc -ffunction-sections -S -x c -o - - \
  | sed -e "s/unused/test/g" \
  | gcc -c -xassembler - -o test
$ objdump -d test | grep .section
Disassembly of section .text.test:
Disassembly of section .text.main:
```

2.3 Link: Link binary objects to target executable

At the linking stage, based on a linker script, the linker should be able to determine which sections should be merged and included to the last executables. When linking, the `-T` option of ld can be used to indicate the path of the linker script, if no such option used, a default linker script is called and can be printed with `ld --verbose`.

Here is a basic linker script:

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SECTIONS
{
  .text           :
  {
    *(.text .stub .text.* .gnu.linkonce.t.*)
    ...
  }
  .data            :
  {
    *(.data .data.* .gnu.linkonce.d.*)
    ...
  }
  /DISCARD/ : { *(.note.GNU-stack) *(.gnu.lto_*) }
}
```

The first two commands tell the target architecture and the ABI, the ENTRY command indicates the entry of the executable and the SECTIONS command deals with sections.

The entry (above is _start, the standard C entry, defined in crt1.o) is the root of the whole executable, all of the other symbols (function or data) referenced (directly or indirectly) by the the entry must be kept in the executable to make ensure the executable run without failure. Besides, the undefined symbols (defined in shared libraries) may also need to be kept with the EXTERN command. and note, the `--entry` and `--undefined` options of ld functions as the same to the ENTRY and EXTERN commands of linker script respectively.

`--gc-sections` will follow the above rule to determine which sections should be reserved and then pass them to the SECTIONS command to do left merging and including. The above linker script merges all section prefixed by .text, .stub and .gnu.linkonce.t to the last .text section, the .data section merging is similar. The left sections will not be merged and kept as their own sections, some of them can be removed by the /DISCARD/ instruction.

Let’s see how `--gc-section` work, firstly, without it:

```
$ echo ’unused(){} main(){}’ | gcc -x c -o test -
$ size test
   text    data     bss     dec     hex filename
    800     252       8    1060     424 test
```

Second, With `--gc-sections` (passed to ld with -Wl option of gcc):

```
$ echo ’unused(){} main(){}’ | gcc -ffunction-sections \
  -Wl,--gc-sections -x c -o test -
$ size test
   text    data     bss     dec     hex filename
    794     244       8    1046     416 test
```

It shows, the size of the .text section is reduced and `--print-gc-sections` proves the dead ‘unused’ function is really removed:

```
$ echo 'unused(){} main(){}' | gcc -ffunction-sections \
-Wl,--gc-sections,--print-gc-sections -x c -o test -
/usr/bin/ld:  Removing unused section '.rodata' in file '.../crt1.o'
/usr/bin/ld:  Removing unused section '.data' in file '.../crt1.o'
/usr/bin/ld:  Removing unused section '.data' in file '.../crtbegin.o'
/usr/bin/ld:  Removing unused section '.text.unused' in file '/tmp/cclR3Mgp.o'
```

The above output also proves why the size of the .data section is also reduced.

But if a section is not referenced (directly or indirectly) by the entry, for instance, if want to put a file into the executable for late accessing, the file can be compressed and put into a .image section like this:

```
...
SECTIONS
{
  ...
  .data            :
  {
    __image_start = .;
    *(.image)
    __image_end = .;
    ...
  }
}
```

The file can be accessed through the pointers: __image_start and __image_end, but the .image section itself is not referenced by anybody, then, `--gc-sections` has no way to know the fact that .image section is used and hence removes .image and as a result, the executable runs without expected, In order to solve this issue, another KEEP instruction of the linker script can give a help.

```
...
SECTIONS
{
  ...
  .data            :
  {
    __image_start = .;
    KEEP(*(.image))
    image_end = .;
    ...
  }
}
```

3 Section garbage collection patchset for Linux

The previous section garbage collection patchset is for the -rc version of 2.6.35, which did add the core support of section garbage collection for Linux but it still has some limitations.

Now, Let's analyze the basic support of section garbage collection patchset for Linux and then list the existing limitations.

3.1 Basic support of gc-sections patchset for Linux

The basic support of gc-sections patchset for Linux includes:

- Avoid naming duplication between the magic sections defined by section attribute instruction and `-ffunction-sections` or `-fdata-sections`

  The kernel has already defined some sections with the section attribute instruction of gcc, the naming method is prefixing the sections with .text., as we have discussed in the above section, the name of the sections may be the same as the ones used by `-ffunction-sections` or `-fdata-sections` and hence break the core rule of section garbage collections.

  Therefore, several patches has been upstreamed to rename the magic sections from {.text.X, .data.X, .bss.X, .rodata.X} to {.text..X, .data..X, .bss..X, .rodata..X} and from {.text.X.Y, .data.X.Y, .bss.X.Y, .rodata.X.Y} to {.text..X..Y, .data..X..Y, .bss..X..Y, .rodata..X..Y}, accordingly, the related headers files, c files, assembly files, linker scripts which reference the sections should be changed to use the new section names.

  As a result, the duplication between the section attribute instruction and `-ffunction-sections`/`-fdata-sections` is eliminated.

- Allow linker scripts to merge the sections generated by `-ffunction-sections` or `-fdata-sections` and prevent them from merging the magic sections

  In order to link the function or data sections generated by `-ffunction-sections` or `-fdata-sections` to the last {.text, .data, .bss, .rodata}, the related linker scripts should be changed to merge the corresponding {.text.*, .data.*, .bss.*, .rodata.*} and to prevent the linker from merging the magic sections(e.g. .data..page_aligned), more restrictive patterns like the following is preferred:

  ```
  *(.text.[A-Za-z0-9_$^]*)
  ```

  A better pattern may be the following:

  ```
  *(.text.[^.]*)
  ```

  Note, both of the above patterns are only supported by the latest ld, please use the versions newer than 2.21.0.20110327 or else, they don't work and will on the contrary to generate bigger kernel image for ever such section will be linked to its own section in the last executable and the size will be increased heavily for the required alignment of every section.

- Support objects with more than 64k sections

  The variant type of section number(the e_shnum member of elf{32,64}_hdr) is _u16, the max number is 65535, the old modpost tool (used to postprocess module symbol) can only handle an object which only has small than 64k sections and hence may fail to handle the kernel image compiled with huge kernel builds (allyesconfig, for example) with -ffunction-sections. Therefore, the modpost tool is fixed to support objects with more than 64k sections by the document "IA-64 gABI Proposal 74: Section Indexes": http://www.codesourcery.com/public/cxx-abi/abi/prop-74-sindex.html.

- Invocation of `-ffunction-sections`/`-fdata-sections` and `--gc-sections`

  In order to have a working kernel with `-ffunction-sections` and `-fdata-sections`:

  ```
  $ make KCFLAGS="-ffunction-sections␣-fdata-sections"
  ```

  Then, in order to also garbage-collect the sections, added

  ```
  LDFLAGS_vmlinux += --gc-sections
  ```

  in the top-level Makefile.

The above support did make a working kernel with section garbage collection on X86 platforms, but still has the following limitations:

1. Lack of test, and is not fully compatible with some main kernel features, such as Ftrace, Kgcov
2. The current usage of section attribute instruction itself still breaks the core rule of section garbage collections for lots of functions or data may be put into the same sections(e.g. __init), which need to be fixed
3. Didn't take care of assembly carefully and therefore, the dead sections in assembly may also be reserved in the last kernel image
4. Didn't focus on the support of compressed kernel images, the dead sections in them may also be reserved in the last compressed kernel image
5. The invocation of the gc-sections requires to pass the gcc options to 'make' through the environment variables, which is not convenient
6. Didn't pay enough attention to the the kernel modules, the kernel modules may also include dead symbols which should be removed
7. Only for X86 platform, not enough for the other popular embedded platforms, such as ARM, MIPS and PowerPC

In order to break through the above limitations, improvement has been added in our gc-sections project, see below section.

3.2 Improvement of the previous gc-sections patchset

Our gc-sections project is also based on mainline 2.6.35(exactly 2.6.35.13), it brings us with the following improvement:

1. Ensure the other kernel features work with gc-sections

   Ftrace requires the __mcount_loc section to store the mcount calling sites; Kgcov requires the .ctors section to do gcov initialization, these two sections are not referenced directly and will be removed by `--gc-sections` and hence should be kept by the KEEP instruction explicitly. Besides, more sections listed in include/asm-generic/vmlinux.lds.h or the other arch specific header files has the similar situation and should be kept explicitly too.

   ```
   /* include/asm-generic/vmlinux.lds.h */
   ...
   -       *(__mcount_loc)          \
   +       KEEP(*(__mcount_loc))    \
   ...
   -       *(.ctors)                \
   +       KEEP(*(.ctors))          \
   ...
   ```

2. The section name defined by section attribute instruction should be unique

   The symbol name should be globally unique (or else gcc will report symbol redefinition), in order to keep every section name unique, it is possible to code the section name with the symbol name. `__FUNCTION__` (or `__func__` in Linux) is available to get function name, but there is no way to get the variable name, which means there is no general method to get the symbol name so instead, another method should be used, that is coding the section name with line number and a file global counter. the combination of these two will minimize the duplication of the section name (but may also exist duplication) and also reduces total size cost of the section names.

gcc provides __LINE__ and __COUNTER__ to get the line number and counter respectively, so, the previous __section() macro can be changed from:

```
#define __section(S) \
  __attribute__ ((__section__(#S)))
```

to the following one:

```
#define __concat(a, b) a##b
#define __unique_impl(a, b) __concat(a, b)
#define __ui(a, b) __unique_impl(a, b)
#define __unique_counter(a) \
  __ui(a, __COUNTER__)
#define __uc(a) __unique_counter(a)
#define __unique_line(a) __ui(a, __LINE__)
#define __ul(a) __unique_line(a)
#define __unique(a) __uc(__ui(__ul(a),l_c))
#define __unique_string(a) \
  __stringify(__unique(a))
#define __us(a) __unique_string(a)

#define __section(S) \
  __attribute__ ((__section__(__us(S.))))
```

Let's use the __init for an example to see the effect, before, the section name is .init.text, all of the functions marked with __init will be put into that section. With the above change, every function will be put into a unique section like .text.init.13l_c16 and make the linker be able to determine which one should be removed.

Similarly, the other macros used the section attribute instruction should be revisited, e.g. __sched.

In order to make the linker link the functions marked with __init to the last .init.text section, the linker scripts must be changed to merge .init.text.* to .init.text. The same change need to be applied to the other sections.

3. Ensure every section name indicated in assembly is unique

`-ffunction-sections` and `-fdata-sections` only works for C files, for assembly files, the .section instruction is used explicitly, by default, the kernel uses the instruction like this: `.section .text`, which will break the core rule of section garbage collection tool, therefore, every assembly file should be revisited.

For the macros, like `LEAF` and `NESTED` used by MIPS, the section name can be uniqued with symbol name:

```
#define LEAF(symbol)    \
-       .section        .text;          \
+       .section        .text.asm.symbol;\
```

But the other directly used .section instructions require a better solution, fortunately, we can use the same method proposed above, that is:

```
#define __asm_section(S) \
  .section __us(S.)
```

Then, every .section instruction used in the assembly files should be changed as following:

```
/* include/linux/init.h */
-#define __HEAD .section        ".head.text","ax"
+#define __HEAD __asm_section(.head.text), "ax"
```

4. Simplify the invocation of the gc-sections

In order to avoid passing `-ffunction-sectoins`, `-fdata-sections` to 'make' in every compiling, both of these two options should be added to the top-level Makefile or the arch specific Makefile directly, and we also need to disable `-ffunction-sectoins` explicitly when Ftrace is enabled for Ftrace requires the `-p` option, which is not compatible with `-ffunction-sectoins`.

Adding them to the Makefile directly may also be better to fix the other potential compatiabilities, for example, `-fdata-sections` doesn't work on 32bit kernel, which can be fixed as following:

```
# arch/mips/Makefile
ifndef CONFIG_FUNCTION_TRACER
  cflags-y := -ffunction-sections
endif
# FIXME: 32bit doesn't␣work␣with␣-fdata-sections
ifdef CONFIG_64BIT
  cflags-y += -fdata-sections
endif
```

Note, some architectures may prefer `KBUILD_CFLAGS` than cflags-y, it depends.

Besides, the `--print-gc-sections` option should be added for debugging, which can help to show the effect of gc-sections or when the kernel doesn't boot with gc-sections, it can help to find out which sections are wrongly removed and hence keep them explicitly.

```
# Makefile
ifeq ($(KBUILD_VERBOSE),1)
LDFLAGS_vmlinux += --print-gc-sections
endif
```

Then, `make V=1` can invocate the linker to print which symbols are removed from the last the executables.

In the future, in order to make the whole gc-sections configurable, 4 new kernel config options may be required to reflect the selection of `-ffunction-sections`, `-fdata-sections`, `--gc-sections` and `--print-gc-sections`.

5. Support compressed kernel image

The compressed kernel image often include a compressed vmlinux and an extra bootstraper, the bootstraper decompress the compressed kernel image and boot it. the bootstraper may also include dead code, but for its Makefile does not inherit the make rules from either the top level Makefile or the Makefile of a specific architecture, therefore, this should be taken care of independently.

Just like we mentioned in section 2.3, the section stored the kernel image must be kept with the KEEP instruction, and the `-ffunction-sectoins`, `-fdata-sections`, `--gc-sections` and `--print-gc-sections` options should also be added for the compiling and linking of the bootstraper.

6. Take care of the kernel modules

   Currently, all of the kernel modules share a common linker script: scripts/module-common.lds, which is not friendly to `--gc-sections` for some architectures may requires to discard some specific sections. therefore, a arch specific module linker script should be added to arch/ARCH/ and the following lines should be added to the top-level Makefile:

```
# Makefile
+ LDS_MODULE       = \
  -T $(srctree)/arch/$(SRCARCH)/module.lds
+ LDFLAGS_MODULE  = \
  $(if $(wildcard arch/$(SRCARCH)/module.lds),\
        $(LDS_MODULE))
LDFLAGS_MODULE  += \
  -T $(srctree)/scripts/module-common.lds
```

   Then, every architecture can add the architecture specific parts to its own module linker script, for example:

```
# arch/mips/module.lds
SECTIONS {
  /DISCARD/ : {
    *(.MIPS.options)
    ...
  }
}
```

   In order to remove the dead code in the kernel modules, it may require to enhance the common module linker script to keep the functions called by module_init() and module_exit(), for these two are the init and exit entries of the modules. Besides, the other specific sections (e.g. .modinfo, __version) may need to be kept explicitly. This idea is not implemented in our gc-sections project yet.

7. Port to the other architectures based platforms

   Our gc-sections have added the gc-sections support for the top 4 architectures (ARM, MIPS, PowerPC and X86) based platforms and all of them have been tested.

   The architecture and platform specific parts are small but need to follow some basic steps to minimize the time cost, the porting steps to a new platform will be covered in the next section.

3.3 The steps of porting gc-sections patchset to a new platform

In order to make gc-sections work on a new platform, the following steps should be followed (use ARM as an example).

1. Prepare the development and testing environment, including real machine(e.g. dev board) or emulator(e.g. qemu), cross-compiler, file system etc.

   For ARM, we choose qemu 0.14.50 as the emulator and versatilepb as the test platform, the corss-compiler (gcc 4.5.2, ld 2.21.0.20110327) is provided by ubuntu 11.04 and the filesystem is installed by debootstrap, the ramfs is available from http://d-i.debian.org/daily-images/armel/.

2. Check whether the GNU toolchains support `-ffunction-sections`, `-fdata-sections` and `--gc-sections`, if no support, add the toolchains support at first

   The following command shows the GNU toolchains of ARM does support gc-sections, or else, there will be failure.

```
$ echo ’unused(){} main(){}’ | arm-linux-gnueabi-gcc \
  -ffunction-sections -Wl,--gc-sections \
  -S -x c -o - - | grep .section
        .section        .text.unused,"ax",%progbits
        .section        .text.main,"ax",%progbits
```

3. Add `-ffunction-sections`, `-fdata-sections`, at proper place in arch or platform specific Makefile

```
# arch/arm/Makefile
ifndef CONFIG_FUNCTION_TRACER
KBUILD_CFLAGS   += -ffunction-sections
endif
KBUILD_CFLAGS   += -fdata-sections
```

4. Fix the potential compatibility problem (e.g. disable `-ffunction-sections` while requires Ftrace)

   The Ftrace compatiability problem is fixed above, no other compatibility has been found up to now.

5. Check if there are sections which are unreferenced but used, keep them

   The following three sections are kept for ARM:

```
# arch/arm/kernel/vmlinux.lds.S
        ...
        KEEP(*(.proc.info.init*))
        ...
        KEEP(*(.arch.info.init*))
        ...
        KEEP(*(.taglist.init*))
```

6. Do basic build and boot test, if boot failure happens, use `make V=1` to find out the wrongly removed sections and keep them explicitly with the KEEP instruction

   ```
   $ qemu-system-arm -M versatilepb -m 128M \
     -kernel vmlinux -initrd initrd.gz    \
     -append "root=/dev/ram␣init=/bin/sh" \
   ```

   If the required sections can not be determined in the above step, it will be found at this step for `make V=1` will tell you which sections may be related to the boot failure.

7. Add support for assembly files with the __asm_section() macro

   Using grep command to find out every .section place and replace it with the __asm_section() macro, for example:

   ```
   # arch/arm/mm/proc-arm926.S
   -       .section ".rodata"
   +       __asm_section(.rodata)
   ```

8. Follow the above steps to add support for compressed kernel

   Enable gc-sections in the Makefile of compressed kernel:

   ```
   # arch/arm/boot/compressed/Makefile
   EXTRA_CFLAGS+=-ffunction-sections -fdata-sections
   ...
   LDFLAGS_vmlinux := --gc-sections
   ifeq ($(KBUILD_VERBOSE),1)
   LDFLAGS_vmlinux += --print-gc-sections
   endif
   ...
   ```

   And then, keep the required sections in the linker script of the compressed kernel:

   ```
   # arch/arm/boot/compressed/vmlinux.lds.in
   KEEP(*(.start))
   KEEP(*(.text))
   KEEP(*(.text.call_kernel))
   ```

9. Make sure the main kernel features (e.g. Ftrace, Kgcov, Perf and Oprofile) work normally with gc-sections

   Validated Ftrace, Kgcov, Perf and Oprofile on ARM platform and found they worked well.

10. Add architecture or platform specific module.lds to remove unwanted sections for the kernel modules

    In order to eliminate the unneeded sections(e.g. .fixup, __ex_table) for modules while no CONFIG_MMU, a new module.lds.S is added for ARM:

    ```
    # arch/arm/module.lds.S
    ...
    SECTIONS {
            /DISCARD/ : {
    #ifndef CONFIG_MMU
                    *(.fixup)
                    *(__ex_table)
    #endif
            }
    }
    # arch/arm/Makefile
    extra-y := module.lds
    ```

11. Do full test: test build, boot with NFS root filesystem, the modules and so forth.

    Enable the network bridge support between qemu and your host machine, open the NFS server on your host, config smc91c111 kernel driver, dhcp and NFS root client, then, boot your kernel with NFS root filesystem to do a full test.

    ```
    $ qemu-system-arm -kernel /path/to/zImage \
      -append "init=/bin/bash␣root=/dev/nfs␣\
    ␣␣nfsroot=$nfs_server:/path/to/rootfs␣ip=dhcp" \
      -M versatilepb -m 128M -net \
      nic,model=smc91c111 -net tap
    ```

4 Testing results

Test has been run on all of the top 4 architectures, including basic boot with ramfs, full boot with NFS root filesystem and the main kernel features (e.g. Ftrace, Kgcov, Perf and Oprofile).

The host machine is thinkpad SL400 with Intel(R) Core(TM)2 Duo CPU T5670, the host sytem is ubuntu 11.04.

The qemu version and the cross compiler for ARM is the same as above section, the cross compiler for MIPS is compiled from buildroot, the cross compiler for PowerPC is downloaded from emdebian.org/debian/, the details are below:

arch	board	net	gcc	ld
ARM	versatilepb	smc91c111	4.5.2	2.21.0.20110327
MIPS	malta	pcnet	4.5.2	2.21
PPC	g3beige	pcnet	4.4.5	2.20.1.20100303
X86	pc-0.14	ne2k_pci	4.5.2	2.21.0.20110327

TABLE 1: *Testing environment*

Note:

- In order to boot qemu-system-ppc on ubuntu 11.04, the openbios-ppc must be downloaded from debian repository and installed, then use the -bios option of qemu-system-ppc to indicate the path of the openbios.

- Due to the regular experssion pattern bug of ld <2.20 described in section 3, In order to make the gc-sections features work with 2.20.1.20100303, the linker script of powerpc is changed through using the pattern .text.*, .data.*, .bss.*, .sbss.*, but to avoid wrongly merging the kernel magic sections (e.g. .data..page_aligned) to the .data section, the magic sections are moved before the merging of .data, then it works well because of the the .data..page_aligned will be linked at first, then, it will not match .data.* and then will not go to the .data section. Due to the unconvenience of this method, the real solution will be forcing the users to use ld >= 2.21, or else, will disable this gc-sections feature to avoid generating bigger kernel image.

```
SECTIONS
{
        ...
        .data..page_aligned : ... {
                PAGE_ALIGNED_DATA(PAGE_SIZE)
        }
        ...
        .data : AT(ADDR(.data) - LOAD_OFFSET) {
                DATA_DATA
                *(.data.*)
                ...
                *(.sdata.*)
                ...
        }
        ...
}
```

The following table shows the size information of the vanilla kernel image(vmlinux, .orig) and the kernel with gc-sections(.gc), both of them are stripped through `strip -x` (or even `strip s`) because of gc-sections may introduce more symbols (especially, non-global symbols) which are not required for running on the embedded platforms.

The kernel config is gc_sections_defconfig placed under arch/ARCH/configs/, it is based on the versatile_defconfig, malta_defconfig, pmac32_defconfig and i386_defconfig respectively, extra config options only include the DHCP, net driver, NFS client and NFS root file system.

arch	text	data	bss	total	save
ARM	3062975	137504	198940	3650762	
	3034990	137120	198688	3608866	-1.14%
MIPS	3952132	220664	134400	4610028	
	3899224	217560	123224	4545436	-1.40%
PPC	5945289	310362	153188	6671729	
	5849879	309326	152920	6560912	-1.66%
X86	2320279	317220	1086632	3668580	
	2206804	311292	498916	3572700	-2.61%

TABLE 2: *Testing results*

5 Conclusions

The testing result shows that gc-sections does eliminate some dead code and reduces the size of the kernel image by about 1˜3%, which is useful to some size critical embedded applications.

Besides, this brings Linux kernel with link time dead code elimination, more dead code can be further eliminated in some specific applications (e.g. only parts of the kernel system calls are required by the target system, finding out the system calls really not used may guide the kernel to eliminate those system calls and their callees), and for safety critical systems, dead code elimination may help to reduce the code validations and reduce the possibinity of execution on unexpected code. And also, it may be possible to scan the kernel modules('make export_report' does help this) and determine which exported kernel symbols are really required, keep them and recompile the kernel may help to only export the required symbols.

Next step is working on the above ideas and firstly will work on application guide system call optimization, which is based on this project and maybe eliminate more dead code.

And at the same time, do more test, clean up the existing patchset, rebase it to the latest stable kernel, then, upstream them.

Everything in this work is open and free, the homepage is tinylab.org/index.php/projects/tinylinux, the project repository is gitorious.org/tinylab/tinylinux.

References

[1] *Tiny Lab, http://www.tinylab.org*

[2] *Link time dead code and data elimination using GNU toolchain, http://elinux.org/images/2/2d/ELC2010-gc-sections_Denys_Vlasenko.pdf*, Denys Vlasenko

[3] *Executable and Linkable Format, http://www.skyfree.org/linux/references/ ELF_Format.pdf*

[4] *GNU Linker ld, http://sourceware.org/binutils/docs-2.19/ld/*

[5] *A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux, http://www.muppetlabs.com/˜breadbox/software/ tiny/teensy.html*

[6] *Understanding ELF using readelf and objdump, http://www.linuxforums.org/articles/understanding-elf-using-readelf-and-objdump_125.html*

[7] *ELF: From The Programmers Perspective, http://www.ru.j-npcs.org/usoft/WWW/www_debian.org/Documentation/elf/elf.html*

[8] *UNIX/LINUX 平台可执行文件格式分析 , https://www.ibm.com/developerworks/cn/linux/l-excutff/*

[9] *Section garbage collection patchset, https://patchwork.kernel.org/project/linux-parisc/list/?submitter=Denys+Vlasenko*, DENYS VLASENKO

[10] *Work on Tiny Linux Kernel, http://elinux.org/Work_on_Tiny_Linux_Kernel*, WU ZHANGJIN