A Fast LKM loader based on SysV ELF hash table

Embedded Linux Conference – Europe 2009

Grenoble, Oct 15th 2009

Carmelo Amoroso

Senior software engineer

STMicroelectronics

carmelo.amoroso@st.com

www.stlinux.com

Agenda

- Fast boot strategies
- Module loader & symbol resolution
- Symbol resolution process
  - Current implementation
  - Simple hash solution
  - Fast SysV ELF hash
- Future work
- Q&A

Speeding up system boot time

- Kernel boot time is a hot topic nowadays for both embedded and desktop systems
- Several solutions proposed
  - Boot-loader speedups
  - User space and applications speedups
  - Specialised filesystems
  - Kernel speedups
    - Uncompressed vmlinux kernel
    - Minimal (tiny and modular) kernel image
    - ...

For further info refer to http://www.elinux.org/Boot_Time

  - *We will focus on kernel module loading process*

Modularised kernel

- Traditionally embedded systems were built using monolithic kernels... but they were:
  - Less complex
  - With less drivers
- Today modular kernels are *"mandatory"* for embedded systems too
  - A lot of drivers and subsystems are built as modules
  - Reducing kernel image size impacts
    - Load and decompression time
    - Allow to fits better the image in flash device
- This is true also for desktop systems
  - 64 modules on my Fedora Core 9 on x86

Modularised kernel

- Modules are loaded
  - After kernel boot from init scripts
  - From initramfs in modern systems
  - On demand via Kmod
- Module load time impacts system performance and boot time globally
- It is another important part of the system that needs to be optimised and upon which we focused

Module loader: *symbol resolution*

- An important step of module loading is **symbol resolution** (*simplify_symbols* function)
- Undefined symbols have to be resolved against exported symbols (kernel and modules)
- Exported symbols are stored in a plain array of struct kernel_symbol
- Such arrays are kept is separate ELF sections ksymtab_{gpl, gpl_future, unused, unused_gpl}

Symbol resolution: *current*

- Symbol look-up is performed by a linear search
  - Matching criteria is simply based on string comparison
  - Never optimised until now
- Cost of symbols resolution for k-th module is
  - $C(k) = O(n + \Sigma\, m(i))$ for $i = 1 .. k-1$
    - Where n is the number of kernel's exported symbols
    - Where m(i) is the number symbols exported by i-th module

kernel: 2.6.23.17 (kernel/module.c)

```
static const struct kernel_symbol *lookup_symbol(const char *name,
                const struct kernel_symbol *start,
                const struct kernel_symbol *stop)
{
        const struct kernel_symbol *ks = start;
        for (; ks < stop; ks++)
                if (strcmp(ks->name, name) == 0)
                return ks;
        return NULL;
}
```

Symbol resolution: *simple hash*

- A first simple optimisation aimed at reducing the number of strcmp calls was implemented
- Struct kernel_symbol extended to include an hash value
- New ELF section .undef.hash added to each module containing the hash values of undefined symbols
  - In the same order as they are listed in the ELF .symtab
  - Created during the modpost stage (in <module>.mod.c)
  - Initially empty (.symtab layout not yet finalised)

Symbol resolution: *simple hash*

- Lookup function modified to call strcmp only when hash values match

Kernel: 2.6.23.17 (kernel/module.c)

```
static const struct kernel_symbol *hash_lookup_symbol(const char *name,
        const unsigned long hash_value,
        const struct kernel_symbol *start,
        const struct kernel_symbol *stop)
{
        const struct kernel_symbol *ks = start;
        for (; ks < stop; ks++) {
        /*
         * If hash values don't match, we are sure symbols are different,
         * otherwise we need to explicitly do string comparison.
        */
        if ((ks->hash_value == hash_value) && (strcmp(ks->name, name) == 0))
                return ks;
        }
        return NULL;
}
```

Symbol resolution: *simple hash*

- Hash values computed by a host tool (ksymhash) after modpost stage
  - For both .undef.hash and each __ksymtab
  - Hash function used is Dan Bernstein's string hash function (same used by binutils for GNU hash table)
  - No runtime overhead for computing hash values
  - Memory footprint
    - 4*n bytes (where n stands for exported symbols no.)
    - 4*m bytes (where m stands for undefined symbols no.)
    - e.g. usbcore.ko
      - 159 undefined symbols
        - » undef.hash is 636 bytes
      - Exports 47 plain symbols, 21 GPL and 3 GPL future
        - » Kernel symbol tables are 284 bytes larger totally
    - **Only 920 bytes larger (over ~130KiB)**

Symbol resolution: *simple hash*

- Placeholder for .undef.hash section has to be created by modpost (in <module>.mod.o)

```
void add_undef_hash(struct buffer *b, struct module *mod) {
        struct symbol *s;
        buf_printf(b, "#ifdef CONFIG_LKM_ELF_HASH\n");
        buf_printf(b, "static unsigned long __symtab_hash[]\n");
        buf_printf(b, "__attribute_used__\n");
        buf_printf(b, "__attribute__((section(\".undef.hash\"))) = {\n");

        for (s = mod->unres; s; s = s->next) {
                /*
                 * Fill with zero, the order of unresolved symbol is not yet correct
                 * This will create a placeholder for the hash values.
                 */
                buf_printf(b, "\t% #8lx,\n", 0L);
        }
        buf_printf(b, "};\n");
        buf_printf(b, "#endif\n");
}
```

Symbol resolution: *simple hash*

- Benchmarks showed a gain of ~70% on average
  - Target: STi7109/SH4-202@266Mhz / rootfs NFS
    - Where gain is ((simple_hash - standard)/standard)*100

Time spent (microsecs) in simplify_symbols function

Module	Standard	Simple hash	Gain %
usbcore	84844	10368	**-88**
ehci_hcd	14841	4151	**-72**
ohci_hcd	11634	3161	**-73**
i2c_core	9011	1770	**-80**
i2c_algo_bit	1707	546	**-68**
jffs2	8587	2580	**-70**
mtdchar	5485	1368	**-75**
mtdblock	3181	1087	**-66**
mtd_blkdevs	4576	1396	**-69**
mtd	1251	479	**-62**

- ***Good but not good enough ;-) !***
- So moved to a full hash table solution

Symbol resolution: *SysV ELF hash*

- Based on older C lib dynamic linker implementation
- No toolchain (binutils) support required/available
- Re-used the simple hash solution too
- Architecture independent
- Available for kernel 2.6.23.17 @ STLinux git
  - http://git.stlinux.com/?p=stm/linux-sh4-2.6.23.y.git
  - Relevant commits are:
    - b3dcbc0d349e0325fef19c8a10974d3c15047eaf
    - a63f1018e474e7eae8ed28dd9109ff4b2a9b0613
    - d3f4d4306ea6095f5b1bef67a51d0be8f4447bec
    - b2756f2303b4dc88e1c354654ee6f4e16e989d7c
  - Port for 2.6.30 series almost completed

Symbol resolution: *SysV ELF hash*

- As for the *"simple hash"* solution we have
  - Kernel_symbol extended to include an hash value

File: include/linux/module.h

```
struct kernel_symbol
{
        unsigned long value;
        const char *name;
#ifdef CONFIG_LKM_ELF_HASH
        unsigned long hash_value;
#endif
};
```

  - New ELF section .undef.hash added
  - Hash values computed by ksymhash at build time

Symbol resolution: *SysV ELF hash*

- A SysV hash table created for each ksymtab by modpost (in <module>.mod.c)
- Same done for kernel (.tmp_vmlinux.mod.c)
- Each hash table placed in separate ELF sections (<ksymtab>.htable)
- Hash table layout can be described by

File: scripts/mod/ksymtable.h

```
/*
 * This maps the ELF hash table
 * The entries in the .hash table always have a size of 32 bits.
 */
struct elf_htable {
        uint32_t nbucket;
        uint32_t nchain;
        uint32_t *elf_buckets;
        uint32_t *chains;
};
```

Symbol resolution: *SysV ELF hash*

- Hash table's bucket length is based on the number of symbols only
  - Computed simply by look-up into a static array
  - Taken from binutils simplest implementation

```
static const size_t elf_buckets[] =
{
  1, 3, 17, 37, 67, 97, 131, 197, 263, 521, 1031, 2053, 4099, 8209,
  16411, 32771, 0
};
```

  - Definitely not optimal
  - Hash table tends to be sparse
  - Acceptable as first solution for modules (few exported symbols)
  - Needs to be addressed for kernel image

Symbol resolution: *SysV ELF hash*

- Lookup function updated accordingly
  - Search performed by a standard hash table look-up
  - Actual strcmp called only if hash values match

Kernel: 2.6.23.17 (kernel/module.c)

```
static const struct kernel_symbol *elf_hash_lookup_symbol(const char *name,
        const unsigned long hash_value, const struct kernel_symbol *start,
        const struct kernel_symbol *stop, const uint32_t *htable) {
        ...[SNIP]...
        idx = hash_value % ht.nbucket;
        for (symidx = ht.elf_buckets[idx]; symidx != -1; symidx = ht.chains[symidx]) {
                ks = start + symidx;
                if ((ks->hash_value == hash_value) && !strcmp(ks->name, name))
                        return ks;
                else
                if (symidx >= ht.nchain)
                        return NULL;
        }
        return NULL;
}
```

Symbol resolution: *SysV ELF hash*

- Hash tables base addresses available to kernel through linker script (__start_<ksymtab>.htable)

File: include/asm-generic/vmlinux.lds.h

```
#define RO_DATA(align)                                          \
        . = ALIGN((align));                                     \
        [SNIP]
        /* Kernel ELF hash symbol table: Normal symbols */          \
        __ksymtab.htable       : AT(ADDR(__ksymtab.htable) - LOAD_OFFSET) {\
        VMLINUX_SYMBOL(__start___ksymtab_htable) = .;       \
        *(__ksymtab.htable)                                 \
        }
        [SNIP]
```

Similarly for all other ksymtable hash tables, that is

__ksymtab_gpl.htable

__ksymtab_gpl_future.htable

__ksymtab_unused.htable

__ksymtab_unused_gpl.htable

Symbol resolution: *SysV ELF hash*

- Example of a modpost output (usbcore.mod.c)
  - usbcore exports 47 plain symbols, 21 GPL and 3 GPL future
    - Value -1 means slot unused

File: drivers/usb/core/usbcore.mod.c

```
#ifdef CONFIG_LKM_ELF_HASH
    #include <linux/types.h>
    static uint32_t htable__ksymtab[]
    __attribute_used__ __attribute__((section("__ksymtab.htable"))) = {
        37, /* bucket length*/
        37, /* chain length */

        /* the buckets */
        16, 36, -1, 7, 35, 34, 14, 17, 37, -1, 8, 10, 22, 13, 18, 3, -1, 29, 44, 26, 0, -1, 15, 2, 32, -1, 4, 33,
            6, -1, 9, 43, 21, 1, -1, 24, 40,

        /* the chains */
        5, 11, -1, 12, -1, -1, 23, -1, -1, 19, -1, 25, 27, 41, -1, 42, -1, -1, -1, 20, 28, 30, 39, -1, -1, -1, -1, -1,
            31, -1, -1, 46, -1, -1, 38, -1, 45,
    };
```

Symbol resolution: *SysV ELF hash*

- Build process at a glance
  1. Build ksymhash host tool
  2. Create vmlinux's hash table
     1. Build temporary vmlinux
     2. Generate hash table source on the fly
     3. Compile it
  3. Link final kernel image
  4. Post-process vmlinux by ksymhash
  5. Modules modpost & post-processing
     1. Generate <module>.mod.c on the fly
     2. Compile it
     3. Link final module <module>.ko
     4. Post-process module by ksymhash

Symbol resolution: *SysV ELF hash*

- Output of kernel build process

```
[amorosoc@stlinux ~/git/linux-sh4-2.6.23.y]make ARCH=sh CROSS_COMPILE=sh4-linux-
  SYMLINK include/asm-sh/cpu -> include/asm-sh/cpu-sh4
... [SNIP] ...
  HOSTCC  scripts/ksymhash/elflib.o
  HOSTCC  scripts/ksymhash/ksymhash.o
  HOSTLD  scripts/ksymhash/ksymhash              <<< Step 1
... [SNIP] ...
  LD      .tmp_vmlinux                           <<< Step 2.1
  HTABLE  .tmp_vmlinux                           <<< Step 2.2
  CC      .tmp_vmlinux.mod.o                     <<< Step 2.3
... [SNIP] ...
  LD      vmlinux.o
  MODPOST vmlinux.o
  LD      vmlinux                                <<< Step 3
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
  SYMHASH vmlinux                                <<< Step 4
  Building modules, stage 2.
  MODPOST 35 modules                             <<< Step 5.1
  CC      drivers/ata/libata.mod.o               <<< Step 5.2
  LD [M]  drivers/ata/libata.ko                  <<< Step 5.3
  SYMHASH drivers/ata/libata.ko                  <<< Step 5.4
... [SNIP] ... the same for all other 34 modules
  CC      sound/stm/snd-stm.mod.o
  LD [M]  sound/stm/snd-stm.ko
  SYMHASH sound/stm/snd-stm.ko
```

1. Build ksymhash host tool
2. Create vmlinux's hash table
   1. Build temporary vmlinux
   2. Generate source file on the fly
   3. Compile it
3. Link final kernel image
4. Post-process vmlinux by ksymhash
5. Modules modpost & post-processing
   1. Generate <module>.mod.c on the fly
   2. Compile it
   3. Link final module <module>.ko
   4. Post-process modules by ksymhash

Symbol resolution: *SysV ELF hash*

- Benchmarks
  - Target: STi7109/SH4-202@266Mhz / rootfs NFS
    - Selected a subset of modules
    - Loaded/removed several times to gather average values
    - Gain is: ((<new> - standard)/standard)*100

Time spent (microsecs) in simplify_symbols

Module	Standard	Simple hash	Simple vs std %	ELF hash	ELF vs Std %
usbcore	84845	10368	-88	843	**-99**
ehci_hcd	14841	4151	-72	635	**-96**
ohci_hcd	11634	3160	-73	499	**-96**
i2c_core	9011	1770	-80	108	**-99**
i2c_algo_bit	1707	546	-68	76	**-96**
jffs2	8587	2580	-70	259	**-97**
mtdchar	5485	1368	-75	96	**-98**
mtdblock	3181	1087	-66	105	**-97**
mtd_blkdevs	4576	1396	-69	103	**-98**
mtd	1251	479	-62	64	**-95**

- ***~96 % faster on average !!!***

Future work

- We're currently working to use GNU hash table
  - Proved to have better performance than SysV ELF hash
  - It uses a Bloom filter to avoid false negatives
    - [Bloom] http://en.wikipedia.org/wiki/Bloom_filter
  - Optimised for cached accesses
  - It is based on work done by Drepper and Jelinek for glibc and bintuils
    - [Jelinek]
      - http://sources.redhat.com/ml/binutils/2006-06/msg00418.html
      - http://sourceware.org/ml/binutils/2006-10/msg00377.html
    - [Drepper]
      - http://people.redhat.com/drepper/dsohowto.pdf
- Other improvements may include
  - Perfect hashing solution
  - ...

Module loader: SysV ELF hash

Thank you for your attention !

Q&A