CAR: Using Cache as RAM in LinuxBIOS

Yinghai Lu AMD. *yinghai.lu@amd.com*
Li-Ta Lo, Gregory R. Watson, Ronald G. Minnich
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, NM 87545
{*ollie, gwatson, rminnich*}*@lanl.gov*

Abstract

LinuxBIOS is fast becoming a widely accepted alternative to the traditional PC BIOS for cluster computing applications and embedded solution. LinuxBIOS is divided into several processing part. 1) auto stage: used to initialize memory and other necessary initialization 2) hardwaremain: used to enumerate devices and allocate resource to them. The auto stage in LinuxBIOS v1 is in assembly language, and in LinuxBIOS v2 it is in C language, but for x86 We depends on ROMCC, it will compile the C code to assembly code, the assembly code will only use registers as local variables, and don't support call and stack. In this paper we will show how to use cache in CPU as RAM before the RAM is initialized. So the auto stage code in C will be compiled with GCC and it will use cache in CPU as stack. The CAR (Cache as RAM) in LinuxBIOS include three parts: cache_as_ram.inc, cache_as_ram_post.c, copy_and_run.c . The cache_as_ram.inc will initialize the cache and use that as stack and it will call real_main in cache_as_ram_auto.c at last; the cache_as_ram_post.c is included into cache_as_ram_auto.c, and it is used to stop the cache as ram after RAM is initlized and use RAM as stack; The copy_and_run.c is included into cache_as_ram_auto.c too. It is used to copy hardwaremain stage code to ram and jmp to it. There is some specific processing about AMD and Intel. Using CAR in LinuxBIOS, we can reduce the size of auto stage, also can use the more complicated code in auto stage that is limited by ROMCC and register. The solution can be used all x86 based platform, and may easier to port to other platform such as MIPS.

1 Introduction

LinuxBIOS [2] is an open-source replacement for the traditional PC BIOS. The PC BIOS was developed in the 1980's for the original IBM PC, and much of the functionality needed to support this legacy hardware still remains in the PC BIOS today. In addition, the vintage operating systems that ran on these machines were dependent on the BIOS for carrying out many of the configuration activities needed for the system to function properly. Modern operating systems are now able to initialize and configure hardware directly, so there is no longer any reason for the BIOS to be involved. The basic principle behind a modern BIOS, like LinuxBIOS, is to do the minimum necessary to enable system hardware, then leave as much device configuration to the operating system as it can. The result of eliminating this unnecessary initialization is a very fast boot time compared to a traditional BIOS.

Another legacy feature provided by the PC BIOS is a 16-bit callback interface using the x86 software interrupt mechanism. However, only a tiny subset of the interface is used by modern operating systems, and in the case of Linux, it is not used at all. LinuxBIOS does not provide this interface, and as a result, is able to substantially reduce its memory footprint. Compared to 256KB required by the PC BIOS, the typical size of LinuxBIOS is just 64KB.

Unlike the PC BIOS, only a very small portion of LinuxBIOS is written in assembly code. For the x86 architecture, this is just enough code to initialize the CPU and switch to 32-bit mode. The rest of LinuxBIOS is written in the C language. This makes LinuxBIOS very portable across different architectures, and it has already been ported to support the Alpha and PowerPC processors.

Using a high-level language also allows LinuxBIOS to employ a much more sophisticated object oriented device model, similar to the one used in the Linux kernel. In such a model, each physical device has a corresponding software object. The object encapsulates information about the physical device and has methods to probe, initialize and configure the device. The main function of LinuxBIOS is really just to organize, query and manage these device objects. This kind of device object model is unheard of in a BIOS implemented in assembly code.

LinuxBIOS has been successfully deployed in a number of real world applications. At Los Alamos National Laboratory, we have Pink and Lightning, two very large production cluster systems that use LinuxBIOS. There are also a number of companies shipping commercial LinuxBIOS-based systems.

LinuxBIOS can be divided into several processing parts. 1) auto stage: used to initialize memory and

other necessary initialization 2) hardwaremain: used to enumerate devices and allocating resource to them. The auto stage in LinuxBIOS v1 is in assembly language, and in LinuxBIOS v2 it is in C language, but for x86 We depends on ROMCC that will compile the C code to assembly code, the assembly code will only use registers as local variables, and don't support call and stack. In following sections of this paper we will show how to use cache in CPU as RAM before the RAM is initialized. So the auto stage code in C will be compiled with GCC and it will use cache in CPU as stack. The CAR (Cache as RAM) support in LinuxBIOS include three parts: cache_as_ram.inc, cache_as_ram_post.c, copy_and_run.c . The cache_as_ram.inc will initialize the cache and use that as stack and it finally call real_main in cache_as_ram_auto.c; the cache_as_ram_post.c is included into cache_as_ram_auto.c, and it is used to stop the cache as ram after RAM is initialized and use RAM as stack; The copy_and_run.c is included into cache_as_ram_auto.c too. It is used to copy hardwaremain stage code to ram and jmp to it. There is some specific processing about AMD and Intel. Using CAR in LinuxBIOS, we can reduce the size of auto stage, also can make the more complicated code in auto stage that is limited by ROMCC and register. The solution can be used all x86 based platform, and maybe more easier to porting to other platform such as MIPS.

2 LinuxBIOS v2 structure and romcc

The LinuxBIOS execution sequences is

a. System booting, the CPU will boot from reset16.inc, it will jump to entry16.inc
b. entry16.inc, will switch to 32-bit protected mode.
c. auto.c, it will be compiled into auto.inc by ROMCC, and auto.inc will only use register as local variables. auto.c main purpose is initialization memory controller or called NorthBridge. For AMD K8 later, it will also initialize the Hypertransport links between CPUS and non-coherent device, that include HT routing table in nodes, and scan HT chains. And do the reset to make width and frequency effective.
d. crt0.S, actually it includes entry16.inc and auto.inc, and will become crt0.s. the crt0.S will copy or unzip the hardwaremain stage code into RAM, and jump to it.
e. The hardwaremain start from c_start.s, and call hardwaremain function.

ROMCC is the critical util that auto.c depends, It will compile auto.c into auto.inc, and it can be used by use MMX and SSE registers to support complicated code. The switch is K8. The ROMCC is done by Eric Biederman. It is used by LinuxBIOS v2 for X86 from June 2003 to June 2005.

3 ROMCC benefits or problems

ROMCC can compile C code to assembly code and the result will only use register as local variables. So we can write the RAM initialization code in C instead of assembly directly. Also for AMD HT code, it would be more useful, because there is a lot of code need to done for HT initialization.

On the other side, there are some problems,

a. the romcc.c is quite long about 25,000 line, and it is not easy maintained. And it all depend on Eric, that is painful for other to help.

b. ROMCC doesn't support stack, so there is no call, and the code will be some big. Esp for the 8 way opteron system.

c. the auto.c must be coded carefully to spare every byte, otherwise it will use out of register. It is painful to make the HT initialization works with limited registers. The HT routing table and HT chain scan need a lot of time to debug to verify if the auto.c coding problem or romcc's bug.

d. for the complicated system, 8 way Opteron System. the auto.c to auto.inc compile time is very long, need 2 or more minutes.

e. the crt0.S in assembly need to be rewrite if we want to port that to other platform

f. If we want to reuse HT initialization code to PowerPC and MIPS, We need have new ROMCC for these platform.

4 cache as ram calling sequences

The LinuxBIOS v2 with CAR enabled execution sequences is

a. System booting, the CPU will boot from reset16.inc, it will jump to entry16.inc
b. entry16.inc, will switch to 32-bit protected mode.
c. cache_as_ram.inc, will initialize the cache in cpu, and prepare that for stack using, and call the real_main in cache_as_ram_auto.c at last.
d. cache_as_ram_auto.c, it will be compiled into auto.inc by gcc, its main purpose is initialization

memory controller or called NorthBridge. For AMD K8 later, it will also initialize the Hypertransport links between CPUS and non-coherent device, that include HT routing table in nodes, and scan HT chains. And do the reset to make width and frequency effective.

e. post_cache_as_ram.c is included in cache_as_ram_auto.c and it is actually with inline assembly languages. It will copy the stack from cache to RAM, and set the new stack , stop the cache_as_ram and also clear the first 1M for hardwaremain. At last it will call copy_and_run.
f. the copy_and_run will copy or unzip hardwaremain in to RAM and jump to it. That is some kind of crt0.S, but is in C.
h. The hardwaremain start from c_start.s, and call hardwaremain function.

Also cache_as_ram_auto.c can be compiled into auto.o directly, together with printk_init.c and vtxprintf.c into init.o to the last image to get more power full printk_debug in auto stage. That can be enabled by setting CONFIG_USE_INIT in MB Config.lb

5 cache_as_ram.inc

cache_as_ram.inc is some 30 lines assembly code to enable cache in CPU, propare that for using stack.

a. enable Variable and Fixed MTRRs

```
/* Set the default memory type and enable fixed and variable MTRRs */
movl    $MTRRdefType_MSR, %ecx
xorl    %edx, %edx
/* Enable Variable and Fixed MTRRs */
movl    $0x00000c00, %eax
wrmsr
```

b. clear MTRRs

c. use FIXED MTRR to enable wanted range in [0xc0000, 0xd0000), the type should be WB (WriteBack) IO.

```
/* enable caching for 16K/8K/4K using fixed mtrr */
movl    $0x269, %ecx  /* fix4k_cf000*/
movl    $0x06000000, %edx /* WB IO type */
xorl    %eax, %eax
wrmsr
```

d. use Variable MTRR to enable cache for ROM

```
/* enable write base caching so we can do execute in place
 * on the flash rom.
 */
movl    $0x202, %ecx
xorl    %edx, %edx
movl    $(XIP_ROM_BASE | MTRR_TYPE_WRBACK), %eax
wrmsr

movl    $0x203, %ecx
movl    $0x0000000f, %edx
movl    $(~(XIP_ROM_SIZE - 1) | 0x800), %eax
wrmsr
```

e. enable the cache through CR0

```
/* enable cache */
movl    %cr0, %eax
andl    $0x9fffffff,%eax
movl    %eax, %cr0
```

f. read the wanted range with LODSL

```
/* Read the range with lodsl*/
movl    $(CacheBase+CacheSize-4), %esi
std
movl    $(CacheSize>>2), %ecx
rep     lodsl
```

g. clear the eanted range with STOSL

```
/* Clear the range */
```

```
movl    $(CacheBase+CacheSize-4), %edi
movl    $(CacheSize>>2), %ecx
xorl    %eax, %eax
rep     stosl
```

h. set the stack pointer to ESP

```
movl    $(CacheBase+CacheSize), %eax
movl    %eax, %esp
```

i. call the cache_as_ram_main

```
/* Restore the BIST result */
movl    %ebp, %eax
/* We need to set ebp ? No need */
movl    %esp, %ebp
pushl %eax  /* bist */
call    cache_as_ram_main
/* We will not go back */
```

6 post_cache_as_ram.c

Post_cache_as_ram will stop the cache_as_ram , also it will clear the RAM if needed.

a. set access to RAM at first with var mtrr

b. copy data from cache to RAM

*memcopy((void *)((CONFIG_LB_MEM_TOPK<<10)-DCACHE_RAM_SIZE), (void *)DCACHE_RAM_BASE, DCACHE_RAM_SIZE); //inline*

to set the new stack in RAM.

c. We need to use

```
__asm__ volatile (
    /* set new esp */ /* before _RAMBASE */
    "subl   %0, %%ebp\n\t"
    "subl   %0, %%esp\n\t"
    ::"a"( (DCACHE_RAM_BASE + DCACHE_RAM_SIZE)- (CONFIG_LB_MEM_TOPK<<10) )
);
```

to set the new stack in RAM.

d. call disable_cache_as_ram

e. call copy_and_run.c will like following

```
        /*copy and execute linuxbios_ram */
        copy_and_run(new_cpu_reset);
        /* We will not return */
    }
    print_debug("should not be here -\r\n");
```

7. disable_cache_as_ram.c

Post_cache_as_ram will stop the cache_as_ram.

a. disable the cache

```
/* We don't need cache as ram for now on */
/* disable cache */
"movl    %cr0, %eax\n\t"
"orl    $(0x1<<30),%eax\n\t"
"movl    %eax, %cr0\n\t"
```

b. clear FIXED MTRR

```
/* clear sth */
"movl    $0x269, %ecx\n\t"  /* fix4k_c8000*/
"xorl    %edx, %edx\n\t"
"xorl    %eax, %eax\n\t"
"wrmsr\n\t"
```

c. enable the cache

```
/* enable cache */
"movl    %cr0, %eax\n\t"
"andl    $0x9fffffff,%eax\n\t"
```

```
"movl   %eax, %cr0\n\t"
"invd\n\t"
```

8 copy_and_run.c

a. find out src and dest for hardwaremain stage code or linuxbios_ram

```
__asm__ volatile (
    "leal  4+_liseg, %0\n\t"
    "leal   _iseg,  %1\n\t"
    : "=a" (src) ,  "=b" (dst)
);
```

b. unzip

the code is from nrv2v.c

c. jmp to hardaremain

```
__asm__ volatile (
    "cli\n\t"
    "leal   _iseg, %edi\n\t"
    "jmp    * %edi\n\t"
);
```

9 cache_as_ram_auto.c

It will include pos_cache_as_ram.c to stop the cache_as_ram and set the new stack and call the final copy_and_run.

9 AMD specific, special MSR about mtrr

For AMD Opteron, We need to set specific MSR to modify the MTRR

```
enable_fixed_mtrr_dram_modify:
    movl    $SYSCFG_MSR, %ecx
    rdmsr
    andl    $(~(SYSCFG_MSR_MtrrFixDramEn|SYSCFG_MSR_MtrrVarDramEn)), %eax
    orl     $SYSCFG_MSR_MtrrFixDramModEn, %eax
    wrmsr
```

12 Goods for CAR

The Car avoid the problems that ROMCC cause, and also make crt0.S to be in C too. The benefits include

a. small code size in auto stage, because We have stack and real call.

b. can code auto stage more easier, don't need to worry about out of registers.

c. compile time get less than 20 seconds.

d. make us more easier to use our code copy_and_run.c and HT related code to other platform such as PowerPC and MIP64.

13 Conclusion

In this paper, we have described CAR (Cache-as-RAM), We can use GCC to auto stage code, can make the code smaller, and will not face the out of registers error in ROMCC, Also can make use to easy port HT support code to PowerPC and MIPS. also can resue copy_and_run.c mostly.

The CAR has been verified in Tyan S2885/S2891/S2895 with AMD Opteron dual core or single core CPU.

We are looking to verify CAR in other X86 platform.

References

[1] AMD. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, May 2005.

[2] Ron Minnich, James Hendricks, and Dale Webster. The Linux BIOS. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[3] Li-Ta Lo, Gregory R. Watson, Ronald G. Minnich. FreeVGA: Architecture Independent Video Graphics Initialization for LinuxBIOS. In *2004 USENIX Annual Techinical Conference*,