Back-tracing in MIPS-based Linux Systems

Kim, Jong-Sung (jsungkim@lge.com)
LG Electronics
Agenda

- Backgrounds
- MIPS stack-frame structure
- Back-tracing in MIPS systems
- Back-tracing from the signal context
- Sample applications
- Summary
- References
- Appendix: Crash Report System applied to LGE products
Backgrounds
The MIPS Core

- Brief history
  - In 1981, a team led by John L. Hennessy at Stanford University started working on what would become the 1st MIPS processor
  - In 1984, Hennessy left Stanford to form MIPS Computer Systems
  - In 1992, SGI bought the company to guarantee the design would not be lost
  - The company became known as MIPS Technologies

- Key concepts
  - Deep instruction pipelines
  - One cycle for one instruction (eliminating interlocks)

- Core design licensing
  - Broadcom (SiByte), IDT, LSI Logic, NEC, Philips, Toshiba, ...

- Very popular in developing CE products (BDP, DTV, PDA, STB, ...)
- Known as rolling back stack-frames is not possible
In many cases, it’s very hard and takes long time to reproduce an error

Just-in-time debug information is very useful

- Process/thread ID
- Register dumps
- Variable dumps
- Programming language-level call-stack
- Et cetera

Back-tracing: extracting the function call-stack
Related Works

- __builtin_return_address function/macro inside GCC
  - Written by Richard Henderson (rth@redhat.com)

- Several just-in-time debug features inside Glibc
  - Including:
    - backtrace(3), backtrace_symbols(3), ...
    - catchsegv(1), libSegFault.so
  - Written by Ulrich Drepper (drepper@redhat.com)

- However, they’re not available for MIPS systems
MIPS Stack-frame Structure
### Conceptual structure of a MIPS stack-frame

<table>
<thead>
<tr>
<th>Base</th>
<th>Offset</th>
<th>Contents</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>old $sp</td>
<td>+0</td>
<td>unspecified</td>
<td>High addresses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>variable size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+16</td>
<td>(if present) incoming arguments passed in stack frame</td>
<td>Previous</td>
</tr>
<tr>
<td>$sp</td>
<td>+0</td>
<td>space for incoming arguments 1-4</td>
<td>Current</td>
</tr>
<tr>
<td></td>
<td></td>
<td>locals and temporaries</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>general register save area</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>floating-point register save area</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>argument build area</td>
<td>Low addresses</td>
</tr>
</tbody>
</table>
Real-world MIPS Stack-frame Structure

- **Sample C function**
  - Nested function
  - Two automatic variables

```c
#include <dlfcn.h>
#include <stdio.h>
...

static int shared_local(void)
{
    void *dl_obj;
    int (*dl_fcn)(void);

    printf("%s\n", __FUNCTION__);
    dl_obj = dlopen("libdynamic.so", RTLD_NOW);
    dl_fcn = (int (*)(void))dlsym(dl_obj, "dynamic_global");
    return dl_fcn();
}
```

- **Stack-frame structure**
  - Reserved region for arguments
  - Old stack-frame pointer
  - Return address

<table>
<thead>
<tr>
<th>Base</th>
<th>Offset</th>
<th>Contents</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>unknown</td>
<td>High addresses</td>
</tr>
<tr>
<td>Old $sp</td>
<td>+36</td>
<td>return address ($ra)</td>
<td>Previous</td>
</tr>
<tr>
<td></td>
<td>+32</td>
<td>old frame pointer ($fp)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+24</td>
<td>local variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+16</td>
<td>local variable</td>
<td></td>
</tr>
<tr>
<td>$sp</td>
<td>+0</td>
<td>not used</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>old context register ($gp)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reserved for argument</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reserved for argument</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reserved for argument</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reserved for argument</td>
<td>Low addresses</td>
</tr>
</tbody>
</table>

- **Hm.. what’s the problem?**
  - Variable offsets from the top of stack
  - This figure is not always true

Great Company Great People
Back-tracing in MIPS Systems
The stack-frame is not enough for back-tracing

- Previous stack-frame pointer
  - Offset from $sp is variable
  - Sometimes not saved
- Return address
  - Offset from $sp is variable
  - Sometimes not saved (but, don’t care in this section)

So, binary code scanning is required to acquire:

- Current stack-frame size
- Offset of stack-stored return address
Function Prologue & Epilogue

- **Prologue for a nested function**
  - Context register setup
  - Current stack-frame allocation
  - Return address saving

```assembly
lui gp, 0x5
addiu gp, gp, -32
add gp, gp, t9
addiu sp, sp, -32
sw ra, 24(sp)
sw gp, 16(sp)
...
```

- **Epilogue for a nested function**
  - Return address loading
  - Current stack-frame de-allocation
  - Function return

```assembly
...  
nop  
lw gp, 16(sp)  
lw ra, 24(sp)  
jr ra  
addiu sp, sp, 32
```
Back-tracing Procedure

- Initialization
  - Registers latching (ra ← $ra, sp ← $sp)
  - Code scanning for current stack-frame size
  - Adjust sp to previous stack-frame (sp ← sp + stack_size)

- Repeat until maximum depth reached or ra is zero
  - Save ra in return address buffer
  - Code scanning for current stack-frame size and offset of saved return address
  - Load return address to ra (ra ← sp[ra_offset])
  - Adjust sp to previous stack-frame (sp ← sp + stack_size)

- Return the count of the return addresses found
## Instruction Formats

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDIU</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: ADDIU rt, rs, immediate
Description: GPR[rt] $\leftarrow$ GPR[rs] + immediate

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: LW rt, offset(base)
Description: GPR[rt] $\leftarrow$ memory[GPR[base] + offset]

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: SW rt, offset(base)
Description: memory[GPR[base] + offset] $\leftarrow$ GPR[rt]
Working source code of backtrace_mips32

```c
#include <stdio.h>

# define abs ( n ) \(( n ) < 0 \ ? \ -( n ) : ( n ) \)

int backtrace_mips32 ( void **buffer, int size )
{
    unsigned long *addr;
    unsigned long *ra;
    unsigned long *sp;
    size_t ra_offset;
    size_t stack_size;
    int depth;

    if (!size)
        return 0;
    if (*buffer || size <= 0)
        return -EINVAL;

    // get current $ra and $sp
    scanf _ volatile _ ( 
        "move $0, %ra\n"
        "move $1, %sp\n"
        : "=x" (ra), "=x" (sp)
    );

    // scanning to find the size of the current stack-frame
    stack_size = 0;
    for (addr = (unsigned long *)backtrace_mips32; !stack_size; ++addr)
    {
        if (*addr & 0xfffff0000 == 0x27bd0000)
            stack_size = abs (short) (*addr & 0xffff);
        else if (*addr == 0x3e0000)
            break;
    }

    sp = (unsigned long *) (unsigned long) sp + stack_size;

    // repeat backward scanning
    for (depth = 0; depth < size & & ra; ++depth)
    {
        buffer[depth] = ra;
        ra_offset = 0;
        stack_size = 0;

        for (addr = ra, !ra_offset || !stack_size; --addr)
        {
            switch (*addr & 0xffff0000)
            {
                case 0x27bd0000:
                    stack_size = abs (short) (*addr & 0xffff);
                    break;

                case 0xaf80000:
                    ra_offset = (short) (*addr & 0xffff);
                    break;

                case 0xb30c0000:
                    return depth + 1;
            default:
                break;
            }
        }

        ra = (unsigned long *) (unsigned long) sp + ra_offset;
        sp = (unsigned long *) (unsigned long) sp + stack_size;
    }

    return depth;
}
```

Great Company Great People

GP248
Back-tracing from The Signal Context
Signal Handler Context

- backtrace_mips32 can’t handle stack-frames from signal contexts
- In the signal handler context:
  - $ra points to the code block (by kernel) in the stack
  - backtrace_mips32 can’t handle this non-function code block

- To back-trace from signal contexts:
  - Skip the kernel-inserted code/data block by referencing the signal context structure (ucontext_t) given to the signal handler
  - Handle the possible leaf function at the top of the call-stack
    - No saved return address
    - No stack-frame
Back-tracing from The Signal Context

- Initialization
  - Find $pc, $ra, and $sp from the signal context structure
    
  (pc ← mcontext_t::pc, ra ← mcontext_t::gregs[31], sp ← mcontext_t::gregs[29])

  - Save pc in return address buffer

  - Code scanning from pc to find stack-frame size and stored ra offset

  - If return address was stored, load it to ra (ra ← sp[ra_offset])

  - Adjust sp to previous stack-frame (sp ← sp + stack_size)

- Repeat until maximum depth reached or ra is zero

  - Save ra in return address buffer

  - Code scanning for current stack-frame size and offset of saved return address

  - Load return address to ra (ra ← sp[ra_offset])

  - Adjust sp to previous stack-frame (sp ← sp + stack_size)

- Return the count of found return addresses
sigbacktrace_mips32 Function

Working source code of sigbacktrace_mips32

```c
#define abs(x) ((x) < 0 ? -(x) : (x))

int sigbacktrace_mips32(void **buffer, int size, ucontext_t const *uc) {
    unsigned long *addr;
    unsigned long *pc, *ra, *sp;
    size_t ra_offset, stack_size;
    int depth;

    if(size == 0)
        return 0;
    if(!buffer || size < 0 || !uc)
        return EINVAL;

    // get current $pc, $ra and $sp
    pc = (unsigned long *)uc->uc_mcontext.pc;
    ra = (unsigned long *)uc->uc_mcontext.uregs[31];
    sp = (unsigned long *)uc->uc_mcontext.uregs[29];

    buffer[0] = pc;
    if(size == 1)
        return 1;

    // scanning to find the size of the current stack-frame
    ra_offset = stack_size = 0;
    for(addr = pc; ra_offset || stack_size; --addr) {
        switch(*addr & 0xffff0000) {
            case 0x27bd0000:
                stack_size = abs((short)*addr & 0xffff);
                break;
            case 0xf8790000:
                ra_offset = (short)*addr & 0xffff;
                break;
            case 0x31c0000:
                goto __out_of_loop;
            default:
                break;
        }
    }

    __out_of_loop:
    if(ra_offset)
        ra = *(unsigned long**)((unsigned long)sp + ra_offset);
    if(stack_size)
        sp = *(unsigned long*)((unsigned long)sp + stack_size);

    // repeat backward scanning
    for(depth = 1; depth < size && ra; ++depth) {
        buffer[depth] = ra;
        ra_offset = stack_size = 0;
        for(addr = ra; !ra_offset || !stack_size; --addr) {
            switch(*addr & 0xffff0000) {
                case 0x27bd0000:
                    stack_size = abs((short)*addr & 0xffff);
                    break;
                case 0xf8790000:
                    ra_offset = (short)*addr & 0xffff;
                    break;
                case 0x31c0000:
                    goto __out_of_loop;
                default:
                    break;
            }
        }
        ra = *(unsigned long**)((unsigned long)sp + ra_offset);
        sp = *(unsigned long*)((unsigned long)sp + stack_size);
    }

    return depth;
}
```

Great Company Great People
More Considerations for The Safer Back-tracing

- Leaf functions
  - Leaf functions usually don’t save registers
  - Leaf functions can run with zero-size stack-frame

- Assembly-coded or hard-optimized functions
  - These functions may not save registers
  - These functions may run with zero-size stack-frame
  - These functions may not have normal function prologue and/or epilogue

- If a function without normal function prologue is located at the first place of a loaded object, sigbacktrace will dereference illegal addresses

- Therefore, back-tracing needs hands of the loaded object/symbol table
Sample Applications
Build & Running Environment

- Processor: Broadcom BCM7440P 266MHz
- Linux kernel: 2.6.12
- C library: uClibc 0.9.28
- GCC version: 3.4.6

- CFLAGS: -O -W -Wall -export-dynamic -fPIC -fno-optimize-sibling-calls -g
Sample Application #1

- Simple application to test backtrace_mips32
  - Using static/shared/dynamic-loaded libraries
  - All functions print its name
  - dynamic_local dumps the call-stack using backtrace_mips32
Outputs from The Application
Outputs from The Application (Stripped Binaries)

```
# ./exe
main
static_global
static_local
shared_global
shared_local
dynamic_global
dynamic_local
/home/jsungkim/tmp/test/libdynamic.so [0x2ac1a45c]
/home/jsungkim/tmp/test/libdynamic.so(dynamic_global + 0x50) [0x2ac1a558]
/home/jsungkim/tmp/test/libshared.so [0x2aab043c]
/home/jsungkim/tmp/test/libshared.so(shared_global + 0x50) [0x2aab04a8]
./exe [0x0040089c]
./exe(static_global + 0x50) [0x00400908]
./exe(main + 0x64) [0x00400814]
/lib/libc.so.0(__uClibc_main + 0x230) [0x2abe354c]
./exe [0x00400674]
```
Outputs from The Application (Optimized Binaries by -O2 or -O3)

```
# ./exe
main
static_global
static_local
shared_global
shared_local
dynamic_global
dynamic_local
/home/jsungkim/tmp/test/libdynamic.so(dynamic_global + 0x8c) [0x2ac1a46c]
/home/jsungkim/tmp/test/libshared.so(shared_global + 0x78) [0x2aaba0428]
./exe(static_global + 0x4c) [0x0040086c]
./exe(main + 0x48) [0x004007f8]
/lib/libc.so.0(_uClibc_main + 0x230) [0x2abe354c]
./exe [0x004000674]
```
Sample Application #2

- Same with sample application #1, except:
  - dynamic_local tries null-pointer assignment
  - sigbacktrace_mips32 is called from the (SIGSEGV handling) signal context
Outputs from The Application
Accompanied to objdump Utility

- If we have binaries compiled with “-g” option...

```c
static int dynamic_local(void)
{
    ...
    printf("%s\n", __FUNCTION__);
    ...
    0320f809 jalr t9
    00000000 nop
    8fde0010 lw gp,16(s8)
    *(unsigned long *)NULL = 0;
    ac000000 sw zero,0(zero)
    return 0;
    00001021 move v0,zero
    03c0e281 move sp,s8
    8fb001c lw ra,28(sp)
    8fbe0018 lw s8,24(sp)
}```
Wrap-up
Summary

- Back-tracing in the MIPS needs some code inspections
- Back-tracing from the signal context needs some more handlings
- Working backtrace/sigbacktrace functions are presented

- Now I’m working on making these functions as an open-source library or inside MIPS-port of C libraries
References

Documents

- MIPS32® Architecture For Programmers - Volume I: Introduction to the MIPS32® Architecture
- MIPS32® Architecture For Programmers - Volume II: The MIPS32® Instruction Set
- System V Application Binary Interface - MIPS® RISC Processor Supplement, 3rd Edition
- Using the GNU Compiler Collection

Internet resources

- MIPS Architecture - History
Appendix: Crash Report System Applied to LGE Products
Crash Report System

- **Purpose**
  - Guarantee not to lose in-time information of system crashes
  - Easy extraction of in-time information
    - `/proc` filesystem entry
    - Extractable to a USB drive

- **With Crash Report System...**
  - All console output is stored on a circular log buffer
  - On watchdog expiration, the captured log is stored to an NVRAM
  - Developers can extract the stored log later
  - The stored log includes the just-in-time debug information
In-time Debug Information

In-time debug information by the sample application