Intricacies of a MIPS Backtrace Implementation

David VomLehn
Technical Leader

16 Feb 2012

© 2012 Cisco and/or its affiliates. All rights reserved.

Stack backtracing and me

Since the beginning, MIPS has really needed a solid binary-only (no symbol table) backtrace library.

In ancient times (early 1990s), porting the C++ thread library to MIPS required a partial stack backtrace implementation, so I wrote one.

As the Technical Committee Chair for the MIPS ABI group in 1995, I drafted a backtrace API for the MIPS ABI Black Book. Unfortunately, that revision of the Black Book has become lost in the mists of time and we're all using an old version.

With over a million Linux-based cable settop boxes deployed and strict limits on down-time, Cisco relies on the ability to remotely diagnose problems. When I realized much more could be done than existing stack backtracing code was doing, I had to do it. [See DSM-IV, Obsessive-Compulsive Disorder]

Today we will discuss backtracing without symbol tables…

© 2012 Cisco and/or its affiliates. All rights reserved.

Naked Stack Backtracing!

© 2012 Cisco and/or its affiliates. All rights reserved.

What is stack backtracing?

Stack backtracing is an iterative process of transforming a state associated with a called function into a state associated with the calling function, until a terminating condition is reached

The state consists of:

- A subset of processor registers--always includes program counter & stack pointer
- A subset of the stack
- Possibly other things, as we will see

Terminating conditions may be:

- Detection that the program counter is within a “root” function function
- A NULL program counter or stack pointer value
- Others as appropriate to the environment

NOTE: in this presentation all stacks are shown with high addresses at the top, low addresses at the bottom

© 2012 Cisco and/or its affiliates. All rights reserved.

Backtracing one frame on an x86 processor

Function prologue:
```
p1: push    ebp
p2: move    ebp, esp
```

Function return:
```
r1: move    esp, ebp
r2: pop     ebp
r3: ret
```

State:

Registers ebp, esp, eip

Stack

Backtrace sequence [offset = sizeof(unsigned long)]:

```
If next instruction to be executed is not at p1
    ESP = EBP
If next instruction to be executed is not at p2, r2, or r3
    EBP = memory[ESP + offset], ESP = ESP + offset
EIP = memory[ESP + offset], ESP = ESP + offset
```

How is MIPS different?

- There is usually no frame pointer
  - If there is a frame pointer, it can be in any of the “saved” registers s0-s8.
  - Once a frame pointer is established, the stack pointer can be modified for things like alloca()
  - The called function will store saved registers before using it, but since it does not know whether the caller is using a saved register as a frame pointers. Thus, frame pointers are not stored at any particular offset in the stack frame
- The return address is placed in the ra register when a call is done (using JAL/JALR opcodes)
  - Leaf functions, i.e. functions that do not call any functions, keep the return address in ra
  - Non-leaf functions can store the return address at any offset within the stack frame
- Stack frame allocated by subtracting the frame size from sp register, deallocated by adding the same value to the sp register
- Leaf functions allocate no stack frames if not needed

© 2012 Cisco and/or its affiliates. All rights reserved.

Why is MIPS different?

- The MIPS calling conventions allow:
  - Increased performance
  - Fewer constraints on the compiler
  - Reduced memory
- Avoiding a dedicated frame pointer:
  - Frees a register to be used instead of memory
- Storing the return address in a register:
  - Avoids the store to and restore from memory for leaf functions
- Decrementing/incrementing the stack pointer instead of pushing/popping a frame pointer:
  - Avoids memory store and restore for a frame pointer
- Avoiding unneeded stack frame allocations eliminates instructions needed to do so

© 2012 Cisco and/or its affiliates. All rights reserved.

MIPS prologue/return code

Function prologue:

```
                jr  ra         ; End of previous function
    <branch delay slot instruction>
p1: addiu   sp, sp, -<frame size>
p3: sw  ra, <ra_offset>(sp)    ; If not a leaf function
    <store any saved registers that will be used on the stack>
p2: move    s<n>, sp       ; If using frame pointer
```

Function return:

```
        r1: move   sp, s<n>       ; If using frame pointer
    <restore any saved registers stored on the stack>
r2: lw  ra, <ra_offset>(sp)    ; If not a leaf function
r3: addiu   sp, sp, <frame size>
r4: jr  ra
```

Notes:

Unlike most processors, there are many possible variants of function prologue and return code sequences and ordering may vary

Function prologue is all in the first basic block

ADDIU SP, SP, <frame size> is normally placed in the JR RA branch delay slot

© 2012 Cisco and/or its affiliates. All rights reserved.

Backtracing one frame on MIPS processors

State:

- Registers s0-s7, s8/fp, sp, ra, pc
- Stack
- Program text
- Other: frame_size, fp_reg_no, is_ra_saved, ra_offset, is_sn_saved[] (true if sn saved), sn_offset[]

Backtrace sequence:

```
SP = REGS[fp_reg_no] + frame_size
if (is_ra_saved)
    RA = memory[SP + ra_offset]
for i in s0, s1, s2, s3, s4, s5, s6, s7, s8/fp, gp
    if (is_sn_saved[i])
    REGS[i] = memory[SP + sn_offset[i]]
```

Encompasses function prologue and return code, but...how to get all those other goodies?

© 2012 Cisco and/or its affiliates. All rights reserved.

Getting the rest of MIPS backtracing state

The basic rules for backtracing a MIPS o32 ABI executable were set out in the 1996 MIPS ABI Black Book (www.sco.com/developers/devspecs/mipsabi.pdf, and others) but it does have some shortcomings:

- Only focused on user space
- Did not handle signals (or exceptions)
- Not concerned with function prologues or returns
- Real gcc code can exit in the middle of a function, without a JR RA to mark the end of the function
- Other optimizations, such as tail recursion, may fall outside the ABI rules

It is based heavily on code analysis

Ultimately must rely on heuristics

Breaks code down into basic blocks:

- Code entered at the top with one exit at the bottom
- Note: this differs in detail from compiler-related usages of this term

© 2012 Cisco and/or its affiliates. All rights reserved.

> “Some debuggers are quite heroic and will even interpret the first few instructions of a function to find how large the stack frame is and to located the stored return address.”

See MIPS Run Linux, Second Edition, Dominic Sweetman

© 2012 Cisco and/or its affiliates. All rights reserved.

ABI-based code analysis

- Scan backwards from the current location:
  - Terminate when one of the following is found:
    - A stack pointer decrement, which will be the beginning of the current function
    - A JR RA instruction, which will be the end of the previous function
  - If a MOVE R<n>, SP (actually, ADDU R<n>, SP, R0) is found:
    - Scan forward to find the JR RA that terminates the function
      - Scan backwards from there looking for MOVE SP, R<m> (actually ADDU SP, R<m>, R0), or the beginning of the last basic block
      - If MOVE SP, R<m> was found and the previous n equals m, set fp_reg_no = n
- If the outermost scan terminated with JR RA:
  - No stack frame was allocated (frame_size = 0, fp_reg_no = 29)
  - No saved registered were stored

© 2012 Cisco and/or its affiliates. All rights reserved.

ABI=based code analysis (con’t)

If the last examined instruction was a stack pointer decrement in the first basic block, this is the stack frame allocation:

- Set frame_size = value of stack pointer decrement

Scan through the first basic block:

- For each instruction storing one of s0-s7, s8/fp, gp relative to the frame pointer (where the stack pointer serves as the frame pointer if none is otherwise used):
  - Set is_sn_saved[register number] = true
  - Set sn_offset[register number] = offset in instruction
- If ra is stored relative to the famer pointer (or stack pointer, if no frame pointer):
- Set is_ra_saved = true
- Set ra_offset = offset in instruction

© 2012 Cisco and/or its affiliates. All rights reserved.

Bounds-based code analysis

- If function bounds are available, can use information to increase robustness
  - Use actual function start instead of using stack frame allocation or JR RA
  - Use actual function end instead of JR RA
- If KALLSYMS is enabled, can use kallsyms_lookup() to get function start and size

© 2012 Cisco and/or its affiliates. All rights reserved.

Making branch delay slots disappear

To handle branch delay slots, one could use:

```
for(;;) {
     rc = fetch_op(pc, &op);
     if (rc != 0)
     return rc;
     if (is_basic_block_end(op))
     break;
     process_op(op);
     pc += 1;
}
rc = fetch_op(pc, &op);
if (rc != 0)
     return rc;
```

Instructions in branch delay slots are properly part of the basic block, so use:

```
for(;;) {
     rc = fetch_op(pc, &op);
     if (rc != 0)
     return rc;
     if (is_basic_block_end(op))
     break;
     process_op(op);
     pc = pc_inc(pc);
}
```

© 2012 Cisco and/or its affiliates. All rights reserved.

Layering on simple stack backtrace

Simple backtrace does one frame.

Looping and adding appropriate termination conditions give lots of functionality

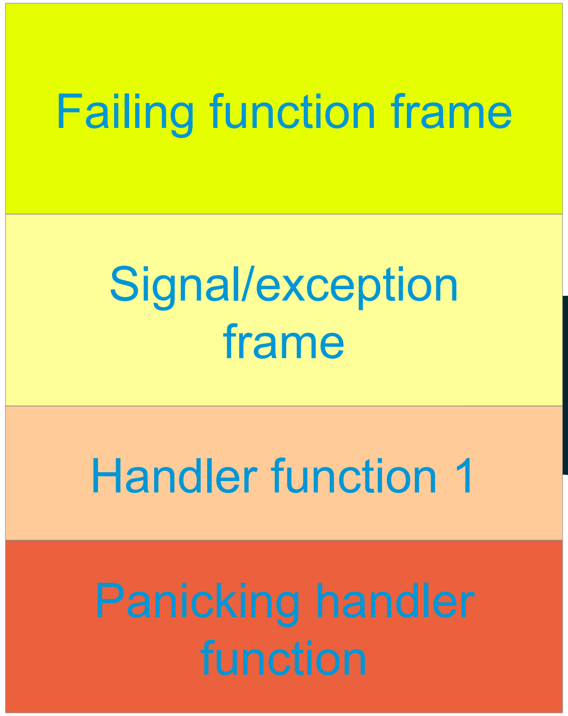
But, wait! There’s more…

- Backtracing over signal frames
- Backtracing over processor exceptions (which, on MIPS, includes interrupts)

© 2012 Cisco and/or its affiliates. All rights reserved.

Why backtrace over signals and exceptions?

signal/exception handlers, but...



← **Want to know information from here**

← **So we can't stop here**

Signal Frames

Detected by matching signal trampoline instruction sequence:

```
li  v0,<value>
syscall
```

This is in the VSDO (virtual dynamically-linked shared object)

Once this has been detected, retrieving the frame’s register values is simply a matter of pulling them off the stack, relative to the stack pointer

© 2012 Cisco and/or its affiliates. All rights reserved.

Exception frames

Detected by program counter within exception handling code

During exception processing, registers are gradually saved within a pt_regs structure by functions like SAVE_SOME, SAVE_ALL, etc.

Uses tables to indicate whether registers values should be taken from the pt_regs structure or the entry registers to the stack backtracing code. Very tedious.

© 2012 Cisco and/or its affiliates. All rights reserved.

Heuristics

Presently rely on two heuristics:

- If not stack frame deallocation found, assume the function doesn't return
- If no return found, use s8 as a potential frame pointer register

Considering heuristic to handle returns from middle of function in ABI backtrace

Other heuristics might help in other places

© 2012 Cisco and/or its affiliates. All rights reserved.

Present usage

- Earlier revision of code is deployed in all Linux-based cable settop boxes
- This stack backtrace used for all kernel backtracing
  - Frame pointer value is available and could be printed to assist in making sense of stack dump
- Also used to backtrace current task, which is stripped, from kernel
  - Task memory map available from /proc
  - Use memory map in conjunction with addr2line and unstripped executable and libraries to print symbolic location
- User space diagnostics code uses older backtrace, intention is to replace with this code

© 2012 Cisco and/or its affiliates. All rights reserved.

Current status

Latest submission to linux-mips mailing list:

- http://comments.gmane.org/gmane.linux.ports.mips.general/31680

Code works well, but Ralf Bachle, MIPS maintainer, doesn’t love a code based approach (and who likes doing all this work?)

Ralf has several requests

- Support Cavium bbit0 instructions
- Create exception handle code table automatically by inserting marks instead of a separate table
- Improve heuristic for missing JR RA in ABI-based code
- Other clean-up

Code undergoing revision for resubmission

© 2012 Cisco and/or its affiliates. All rights reserved.

The future…

Sure would be nice to dispense with code analysis, but…

    - Symbol tables (derived from DWARF symbol table) are big!
    - Could there be a nice hash algorithm compactly mapping function addresses to tables of frame pointer register number, stack frame size, etc., to be computed for each build?

Support other ABIs: n32, n64

- Probably not too difficult
- Probably want to support multiple ABIs simultaneously

MicroMIPS support:

- Need to update instruction decoding, which should be straight-forward
- Need to track the mode

© 2012 Cisco and/or its affiliates. All rights reserved.

Thank you.

Supporting Inline Material

© 2012 Cisco and/or its affiliates. All rights reserved.

SYSTEM V APPLICATION BINARY INTERFACE: MIPS

The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, California, 95060, USA

Stack Backtracing

There are standard called function rules for functions that allocate a stack frame and because the operating system kernel initializes the return address register *$31* to zero when starting a user program it is possible to trace back through any arbitrarily nested function calls. The following algorithm, which takes the set of general registers plus the program counter as input, produces the values the registers had at the most recent function call. Of course, only the saved registers plus *gp, sp, ra,* and *pc* can be reconstructed.

- Scan each instruction starting at the current program counter, going back- wards. The compiler and linker must guarantee that a jump register to re- turn address instruction will always precede each text section.

© 2012 Cisco and/or its affiliates. All rights reserved.

MIPS ABI (2 of 4)

  - If the instruction is of the form “move *$r, sp”* or *“addu $r, $sp, $0,* then the register *$r* may be a frame pointer. The algorithm remembers the current instruction so it can continue its backward scan. Then, it scans forward until it sees the “jr *ra”* instruction that marks the end of the current function.

    Next, it scans backwards searching for an instruction of the form ”move *sp, $r”* or “addu *$sp, $r, $0”.* This scan terminates when such an instruction is found or the branch or jump instruction that marks the beginning of the last basic block.

    If a move or addu instruction of the kind described above was found, remember the register number of *$r* as the frame pointer. Otherwise, *$r* is not the frame pointer.

    The algorithm should return to its original backwards scan starting with the instruction preceding the one remembered above.

- If the instruction is a stack pointer decrement, exit the scan.
- If the instruction is a jump register to return address, exit the scan.

© 2012 Cisco and/or its affiliates. All rights reserved.

MIPS ABI (3 of 4)

- If the last examined instruction is a jump register to the return address, it is the end of the previous function and no stack frame has yet been allocated for the current function. The address from which the current function was called is in the return address register minus eight. The other save registers had their current values when this function was called, so just return their current values.
- The stack decrement instruction must occur in the first basic block of the function. The amount of stack decrement is the size of the stack frame.
- Examine each instruction at increasing program addresses. If any instruction is a store of save registers *$16-$23, $28, $30,* or *$31* through the frame pointer (or stack pointer if no frame pointer was used), then record its value by reading from the stack frame.
- Stop after examining the instruction in the first branch delay slot encountered. This marks the end of the first basic block.

© 2012 Cisco and/or its affiliates. All rights reserved.

MIPS ABI (4 of 4)

- The frame pointer is the stack pointer value at the time the current function was called (or the stack pointer if no frame pointer was used) plus the size of the stack frame.
- The address from which the function is called is either the return address register value minus eight or, if the return address was saved on the stack, the saved value minus eight.

□ 1990-1996 The Santa Cruz Operation, Inc. All rights reserved.