Some new[1] tricks for better performance in MIPS-Linux

David Daney
(Cavium Networks)

1: And some slightly older.

Topics

- User space optimizations
  - MIPS overview
  - Trick 1: Function Prolog optimization
  - Trick 2: Direct call with -mno-shared
  - Trick 3: Non-PIC executables
  - Benchmark results
- Kernel optimizations
  - Kernel ABI
  - Trick 4: Compilation with -msym32
  - Trick 5: Mapped Kernel
- Trick 6: GDB hardware watchpoints

16-bit immediate data

- Property of most RISC architectures
- Multiple instructions are needed to generate constants wider than 16 bits.
- No direct addressing is available. All memory addresses must be loaded into registers to be used.

```
00000000 <foo>:
   0:   3c026655    lui v0,0x6655
   4:   34424433    ori v0,v0,0x4433
   8:   03e00008    jr  ra
   c:   8c420000    lw  v0,0(v0)
```

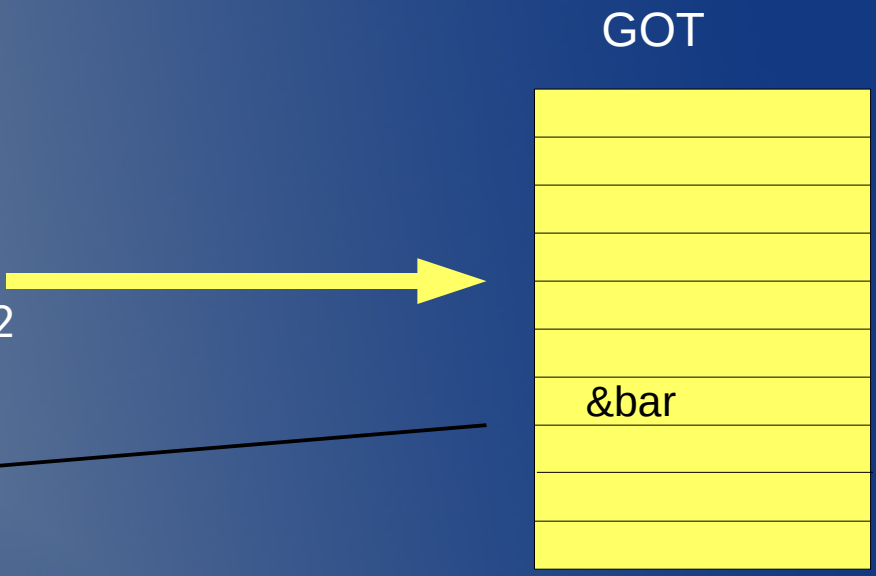
```
int foo()
{
  return *(int *)0x66554433;
}
```

Standard Linux user-space ABIs

- Application Binary Interface (ABI): The rules governing function calls and linking together independent modules.
  - For MIPS there are three (o32, n32, n64)
- Standard ABIs are position independent.
  - Needed for shared libraries.
- Use Global Offset Table (GOT).
  - Function prolog must initialize GOT pointer.
  - Fewer instructions needed to load addresses.
  - Faster runtime linking of shared libraries.
  - Indirect function call.

Function prolog (PIC)

```
00000000 <bam>:
   0:   3c1c0000    lui gp,0x0
            0: R_MIPS_HI16 _gp_disp
   4:   279c0000    addiu   gp,gp,0
            4: R_MIPS_LO16 _gp_disp
   8:   0399e021    addu    gp,gp,t9
   c:   27bdffe0    addiu   sp,sp,-32
  10:   afbf001c    sw  ra,28(sp)
  14:   afbc0010    sw  gp,16(sp)
  18:   8f990000    lw  t9,0(gp)
            18: R_MIPS_CALL16  bar
  1c:   0320f809    jalr    t9
  20:   00000000    nop
  24:   8fbf001c    lw  ra,28(sp)
  28:   8fbc0010    lw  gp,16(sp)
  2c:   24420003    addiu   v0,v0,3
  30:   03e00008    jr  ra
  34:   27bd0020    addiu   sp,sp,32
```




```
int bar();

int bam()
{
  return bar() + 3;
}
```

_gp_disp = &GOT - &bam

Register t9 = &bam

Trick 1: Prolog optimization (non-PIC)

```
00000000 <bam>:
   0:   3c1c0000    lui gp,0x0
            0: R_MIPS_HI16 __gnu_local_gp
   4:   27bdffe0    addiu   sp,sp,-32
   8:   279c0000    addiu   gp,gp,0
            8: R_MIPS_LO16 __gnu_local_gp
   c:   afbf001c    sw  ra,28(sp)
  10:   afbc0010    sw  gp,16(sp)
  14:   8f990000    lw  t9,0(gp)
            14: R_MIPS_CALL16  bar
  18:   0320f809    jalr    t9
  1c:   00000000    nop
  20:   8fbf001c    lw  ra,28(sp)
  24:   8fbc0010    lw  gp,16(sp)
  28:   24420003    addiu   v0,v0,3
  2c:   03e00008    jr  ra
  30:   27bd0020    addiu   sp,sp,32
```

__gnu_local_gp = &GOT

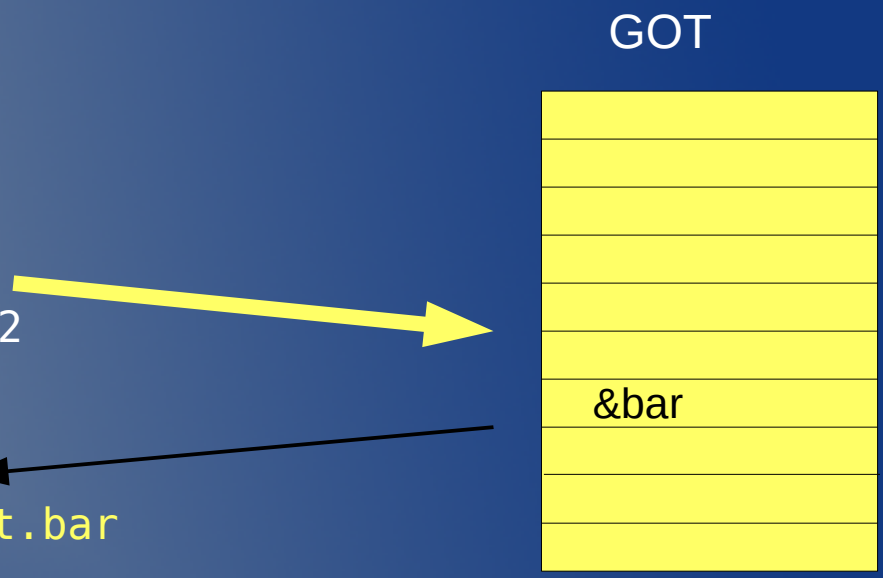
Prolog is now only two instructions. Better instruction scheduling. GNU extension to the standard ABI.

-mno-shared GCC/gas switch

- GNU Binutils 2.16 or newer required for __gnu_local_gp support in ld (linker).
- GCC-4.2: Use -mno-shared for prolog optimization in non-PIC code.
- GCC-4.3: -mno-shared becomes the default for non-PIC code.
- GCC-4.1 and older: Use -Wa,-mno-shared to pass the option to gas.

Indirect function calls

```
00000000 <foo>:
   0:   3c1c0000    lui gp,0x0
            0: R_MIPS_HI16 _gp_disp
   4:   279c0000    addiu   gp,gp,0
            4: R_MIPS_LO16 _gp_disp
   8:   0399e021    addu    gp,gp,t9
   c:   27bdffe0    addiu   sp,sp,-32
  10:   afbf001c    sw  ra,28(sp)
  14:   afbc0010    sw  gp,16(sp)
  18:   8f990000    lw  t9,0(gp)
            18: R_MIPS_GOT16    .text.bar
  1c:   27390000    addiu   t9,t9,0
            1c: R_MIPS_LO16     .text.bar
  20:   0320f809    jalr    t9
  24:   00000000    nop
  28:   8fbf001c    lw  ra,28(sp)
  2c:   24030035    li  v1,53
  30:   70431002    mul v0,v0,v1
  34:   03e00008    jr  ra
  38:   27bd0020    addiu   sp,sp,32
```



Extra instruction(s) plus memory read required to load function address into register.

```
static int bar(int a, int b)
{
  return a - b;
}

int foo(int a, int b)
{
  return bar(a, b) * 53;
}
```

Trick 2: Direct call with -mno-shared

```
00000000 <foo>:
   0:   27bdffe0    addiu   sp,sp,-32
   4:   afbf001c    sw  ra,28(sp)
   8:   0c000000    jal 0 <foo>
            8: R_MIPS_26    .text.bar
   c:   00000000    nop
  10:   8fbf001c    lw  ra,28(sp)
  14:   24030035    li  v1,53
  18:   70431002    mul v0,v0,v1
  1c:   03e00008    jr  ra
  20:   27bd0020    addiu   sp,sp,32
```

Single instruction call. No GOT read. In fact the GOT is not referenced at all so there is no need to initialize gp.

No need to initialize t9 with target function address as it is known to use two instruction optimized prolog sequence.

Cannot be used to call functions outside of the compilation unit as it is not known if they are close enough, or use the optimized prolog sequence.

Trick 3: -mplt (non-pic executables)

- Not quite available yet
  - New in GCC-4.4, Binutils 2.19, glibc 2.9
- Uses -mno-shared like code for all calls
- Not used in shared libraries.
- Not used with n64 ABI.
- Linker generates shim code (PLT stub) for shared library calls.
- 5% faster on common benchmarks.
- Can be slower in some cases
  - Programs that only call library functions.

CSiBE benchmark results

Compiled code size of 893 open-source C files

	.text	% of 3.4.3	.text + .data +.bss	% of 3.4.3
GCC-3.4.3 -Os	4528676	100.00%	5406444	100.00%
GCC-4.4 -Os -mshared	4291772	94.77%	5182092	95.85%
GCC-4.4 -Os -mno-shared	4190908	92.54%	5081220	93.98%
GCC-4.4 -Os -mplt	3714144	82.01%	4596124	85.01%

64-bit Kernel addresses

- Linux kernel ABI does not use a GOT
  - 2 instructions to load a pointer or access global data in 32-bit kernel
  - 6 instructions to load a pointer or access global data in 64-bit kernel, unless...
  - -msym32 gives the same code size in 64-bit kernel as the 32-bit case.

Trick 4: Kernel ABI compilation (-msym32)

-mabi=64 -mno-abicalls -fno-pic

-msym32

```
0000000000000000 <get_c>:
   0:   3c030000    lui v1,0x0
            0: R_MIPS_HI16 c
   4:   03e00008    jr  ra
   8:   8c620000    lw  v0,0(v1)
            8: R_MIPS_LO16 c
```

```
0000000000000000 <get_c>:
   0:   3c030000    lui v1,0x0
            0: R_MIPS_HIGHEST  c
   4:   3c020000    lui v0,0x0
            4: R_MIPS_HI16 c
   8:   64630000    daddiu  v1,v1,0
            8: R_MIPS_HIGHER   c
   c:   0003183c    dsll32  v1,v1,0x0
  10:   0062182d    daddu   v1,v1,v0
  14:   03e00008    jr  ra
  18:   8c620000    lw  v0,0(v1)
            18: R_MIPS_LO16    c
```

```
extern int c;
int get_c(void)
{
  return c;
}
```

-msym32 is only usable if it is known at compile time that the address is in the range 0xffffffff80000000 – 0xffffffffffffffff as in this range the lower 32 bits are always properly sign extended. Actually it works for the range 0 – 0x7fffffff too, but that range is not used by the kernel.

-msym32 Switch

- First available in GCC-4.0.
- Previous to GCC-4.0 kernel had 'hacks' to achieve similar code, but they don't work with 4.0.

256MB range direct jumps

- Linux kernel ABI does not use a GOT.
  - Function calls are direct.
    - Single instruction.
  - The kernel typically resides in KSEG0
    - 0x8000000 (32-bit) or 0xffffffff80000000 (64-bit)
- Kernel modules loaded 'far' from kernel.
  - Modules are typically loaded in SSEG
    - 0xc0000000 (32-bit) or 0xfffffffc0000000 (64-bit)
- Function calls in modules must be indirect to reach kernel.
  - 3 instructions in 32-bit kernel
  - 3 instructions in 64-bit kernel with -msym32
  - 7 instructions in 64-bit kernel without -msym32

Kernel function calls

Core kernel

```
0000000000000000 <do_call>:
   0:   67bdfff0    daddiu  sp,sp,-16
   4:   ffbf0008    sd  ra,8(sp)
   8:   0c000000    jal 0 <do_call>
                8: R_MIPS_26    f
   c:   00000000    nop
  10:   0000102d    move    v0,zero
  14:   dfbf0008    ld  ra,8(sp)
  18:   03e00008    jr  ra
  1c:   67bd0010    daddiu  sp,sp,16
```

```
extern void f(void);
int do_call(void)
{
  f();
  return 0;
}
```

In kernel modules
(32-bit or -msym32 and -mlong-calls)

```
0000000000000000 <do_call>:
   0:   67bdfff0    daddiu  sp,sp,-16
   4:   3c020000    lui v0,0x0
                4: R_MIPS_HI16 f
   8:   ffbf0008    sd  ra,8(sp)
   c:   64420000    daddiu  v0,v0,0
                c: R_MIPS_LO16 f
  10:   0040f809    jalr    v0
  14:   00000000    nop
  18:   dfbf0008    ld  ra,8(sp)
  1c:   0000102d    move    v0,zero
  20:   03e00008    jr  ra
  24:   67bd0010    daddiu  sp,sp,16
```

Trick 5: Mapped Kernel

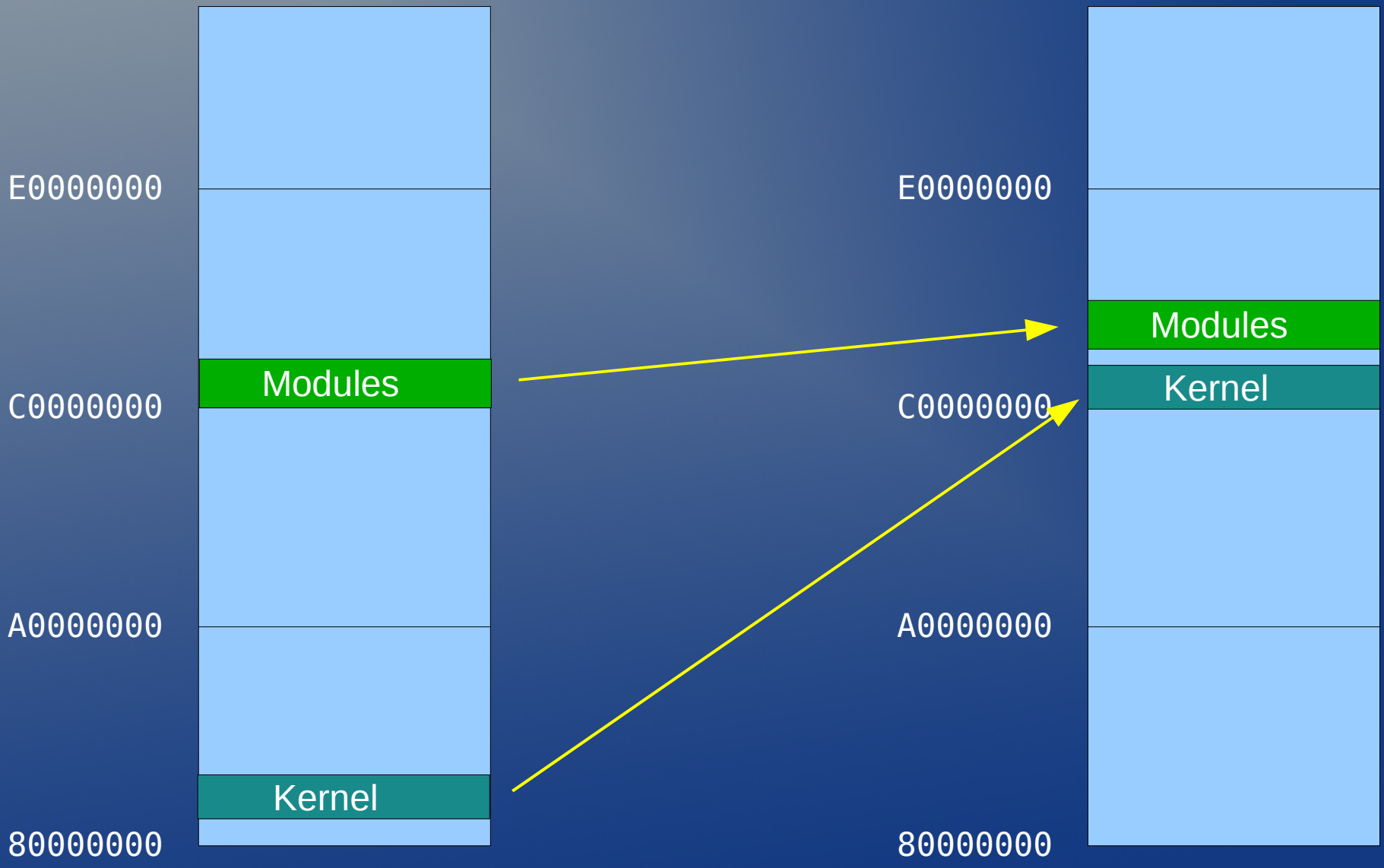
- Move the kernel to sseg
- “Close” to modules
  - Single instruction direct function call.
- Kernel uses a TLB entry
  - Could increase TLB pressure.

Patch: http://www.linux-mips.org/archives/linux-mips/2009-01/msg00010.html

Normal and mapped layouts

Normal kernel. Modules too far from kernel for direct calls.

Mapped kernel. Modules close to kernel, direct calls are possible.



Physical and Virtual addresses differ

- Modify vmlinux.lds to specify separate load address.
- Change kernel address in Makefile
- Bootloader must support loading kernel to proper physical address when it differs from virtual address.
- Modify __pa_symbol() macro.

Kernel now far from exception vectors

- Modify exception vectors to reach the kernel with indirect jump for dedicated interrupt vector.
- All other TLB and exception handlers unchanged.
  - TLB refill never calls the kernel.
  - Exception handlers are via a jump table.

Move module space up.

- Kernel now occupies lowest part of sseg.
- Module memory allocation must be moved above kernel mapping.

Set kernel TLB entry.

- A single TLB entry is used
  - Index 0.
- “Large” pages to cover kernel with single TLB entry
- Communicate end of kernel mapping to module allocator.
- Don't clobber wired value in tlb_init()
- Prototype implementation limitations
  - Arbitrary page size
  - 64-bit kernel

Mapped kernel benchmark

Ethernet device driver (cavium-ethernet) and ipv6 modules

	'Normal' kernel	Mapped kernel	Change
Forwarding IPv6	656000 pkt/s	688000 pkt/s	4.8% better
Ipv6 Module size	282948 bytes	261592 bytes	7.5% smaller

Trick 6: Hardware watch register support for user-space debugging

- GDB can find “memory clobbers” in real time.
  - Software watch points use single stepping and are rarely usable.
    - Extremely slow.
    - Gets stuck looping forever in synchronization primitives.
- Kernel support present in 2.6.28, bug fixes in 2.6.29.
- GDB patchs necessary:

http://sourceware.org/ml/gdb-patches/2009-04/msg00102.html

http://sourceware.org/ml/gdb-patches/2009-04/msg00103.html

New ptrace methods.

- GDB queries kernel for number and size of watch registers.
- GDB sets watch register values.
- When target program traps, GDB queries status of watch registers to find out what happened.

Extensible ptrace interface

```
enum pt_watch_style {
    pt_watch_style_mips32,
    pt_watch_style_mips64
};
struct mips32_watch_regs {
    unsigned int watchlo[8];
    /* Lower 16 bits of watchhi. */
    unsigned short watchhi[8];
    /* Valid mask and I R W bits.
     * bit 0 -- 1 if W bit is usable.
     * bit 1 -- 1 if R bit is usable.
     * bit 2 -- 1 if I bit is usable.
     * bits 3 - 11 -- Valid watchhi mask bits.
     */
    unsigned short watch_masks[8];
    unsigned int num_valid;
} __attribute__((aligned(8)));

struct pt_watch_regs {
    enum pt_watch_style style;
    union {
        struct mips32_watch_regs mips32;
        struct mips64_watch_regs mips64;
    };
};
#define PTRACE_GET_WATCH_REGS 0xd0
#define PTRACE_SET_WATCH_REGS 0xd1
```

```
struct mips64_watch_regs {
    unsigned long long watchlo[8];
    unsigned short watchhi[8];
    unsigned short watch_masks[8];
    unsigned int num_valid;
} __attribute__((aligned(8)));
```

Kernel overhead for watchpoints

- Very low overhead for non-traced tasks.
  - 3 instructions on task switch.
- Watch registers are loaded from thread_struct on task switch.
- No need to clear watch registers when switching away from traced task.
  - Spurious watch traps are ignored.

GDB example session

```
# ./gdb ./watchtest
GNU gdb (GDB) 6.8.50.20090404-cvs
Copyright (C) 2009 Free Software Foundation, Inc.
[...]
(gdb) attach 751
Attaching to program: /junk/watchtest, process 751
[...]
0x2e62d678 in read () from /lib/libc.so.6
0x2e62d678 <read+36>:   bnez    a3,0x2e62d620
(gdb) watch fa[37].b
Hardware watchpoint 1: fa[37].b
(gdb) c
Continuing.
[New Thread 0x2b2cc4d0 (LWP 757)]
[Switching to Thread 0x2b2cc4d0 (LWP 757)]
Hardware watchpoint 1: fa[37].b

Old value = 0
New value = 40
0x00400840 in worker_thread (arg=0x0) at watchtest.c:30
30              fa[i].b = i + 3;
(gdb) c
Continuing.
[Thread 0x2b2cc4d0 (LWP 757) exited]

Program exited normally.
```