A new ABI for little-endian PowerPC64
Design & Implementation

Dr. Ulrich Weigand
Senior Technical Staff Member
GNU/Linux Compilers & Toolchain

Date: Apr 8, 2014
Agenda

• The little-endian PowerPC64 platform
• Goals & methods of ABI design
• Overview of the new ABI
  – In-depth: Establishing TOC addressability
  – In-depth: Passing parameters in memory vs. register
• Implementation status
• Observations on ABI implementation in LLVM
Contributors

Michael Gschwind
Ulrich Weigand
Steve Munroe
David Edelsohn
Alan Modra
Bill Schmidt
Anton Blanchard
Mike Meissner
Ian McIntosh
Julian Wang
The little-endian PowerPC64 platform
Power: big-endian vs. little-endian

• Status of endian support in the past
  – Power ISA has long supported both BE and LE
  – Actual Power hardware/firmware support for LE weak
  – 64-bit server OSes always were BE only

• What is changing?
  – Power8 HW/FW will fully support LE
  – Power LE Linux distributions in development

• Why this change now?
  – Customer requests to simplify application porting and access to certain hardware extensions
  – Tied into the OpenPOWER Foundation effort
Power LE Linux

• How will Linux support Power LE?
  – New architecture `powerpc64le-ibm-linux`
  – No “multilib” co-existence support planned
  – Support for 64-bit applications only
  – Supported only on Power8 and up
  – Linux distribution support to be announced

• What changes are required in Linux?
  – Byte order – obviously
  – New ABI – since we have the opportunity
  – Otherwise, just another platform
Designing a new ABI for PowerPC64
Goals & Methods
PowerPC ABI – current status

• **Current PowerPC ABI conceived over 25 years ago**
  – Reflects hardware implementations tradeoffs
    • E.g., single chip vs. multi-chip implementation
  – Reflects programming usage evolution and paradigms
    • E.g., FORTRAN vs. object oriented programming
    • E.g., lexical nesting rarely used in current languages

• **Opportunity to introduce changes now**
  – Other platforms have introduced new ABIs with 64bit
  – Only incremental improvements on POWER so far
    • Could not break compatibility!
  – Exploit new hardware capabilities
    • Fusion; Improved indirect branch performance
New Power Linux ABI design goals

• **Starting point: PPC64 / AIX ABI**
  – Established, tested production code
  – Leverage commonality across LE, BE and AIX
  – Minimum disruption for tooling

• **Define new capabilities as delta over baseline**
  – Align with the Intel ecosystem
  – Create hardware optimization opportunities / synergies
  – Optimize for modern code patterns
    • More classes, abstraction
    • Shorter function lengths
    • More indirect calls

• **If it ain’t broken, don’t fix it!**
Design approach

• **Compatible implementation**
  – ELFv1 vs. ELFv2 orthogonal to LE vs. BE
  – Full support for ELFv2 testing on BE hardware/OS

• **Hands-on prototyping**
  – Prototype ABI variants through core toolchain stack
    • Binutils, GCC, glibc, set of core libraries
  – Support execution of variant-ABI executables
    • Per-ABI ELF interpreter paths; co-installable

• **Full-scale benchmarking**
  – Build all of SPECint, SPECfp, Python2/3 benchmarks
  – Evaluate actual performance numbers on real hardware
Overview of the PowerPC64 ELFv2 ABI
ELFv2 ABI: Key improvements

• **Execution without functions descriptors**
  – Use of dual entry points to reduce local call cost
  – TOC base materialization using non-PIC and PIC code

• **Optimize for main**
  – Main module to be built without PIC code
  – Symbols in main not dynamically resolved

• **Parameter passing**
  – Pass/return more structures in registers

• **Streamline stack frame**
  – Allocate parameter save area only when required
  – Drop unused words
ELFv2 ABI: Best practices as default

- **Optimize function cross-module calls**
  - Scheduled GOT pointer save in caller
  - Option to inline PLT stub
- **“Medium Code Model” as default**
  - Avoid TOC overflow code
  - Leverage Fusion capability in Power8
- **More descriptive object file info**
  - More precise DWARF, Reloc’s, and ELF format flags
  - Improve future ABI extensibility
In-depth: Establishing TOC addressability
Background: TOC pointer

- The TOC pointer (GOT pointer) is a value that points to a data dictionary and/or the data
  - On 64-bit Power this value is stored in r2
- Data can be addressed either
  - by loading the address of data from the TOC (GOT) and then using the address to so loaded to access data (TOC/GOT-indirect)
  - by loading data from the TOC (TOC-relative)
- Each module has a different TOC
  - Cross-module calls must save and restore old TOC, and load appropriate new TOC value
Background: Function calls

- **Direct calls refer to function symbol**
  - Resolved at link time to target function address if known local (in the same module)
  - Resolved to linker-generated PLT stub if possibly global (in another module)
  - Dynamic loader redirects PLT to final target

- **Indirect calls refer to variable holding a target address**
  - Used to implement C function pointers, C++ virtual functions etc.
Determine new TOC value

• Various options used in other ABIs
  – Caller: Provide TOC value to callee
    • Easy if local call; complicated if not
    • Implemented via function descriptors on Power
  – Callee: Load TOC value as absolute address
    • Prevents position-independent code
  – Callee: Compute TOC value based on current code load address – need to determine that address!
    • Via PC-relative instructions if available (not on Power)
    • Via an artificial “function call” (expensive)
    • Provided by caller (may prevent use of direct calls)
Solution chosen for ELFv2 ABI

• Two entry points for each function:
  – Local EP: TOC expected in r2
  – Global EP: EP address expected in r12
    • Prologue code computes TOC from EP address
    – Just one single ELF symbol (points to global EP)
      • Delta to local EP encoded in ELF st_other bits

• Call sequences:
  – Direct call provides current TOC (already in r2)
    • If known local at link time, call resolved to Local EP
    • If redirected to PLT stub, stub loads target Global EP address from TOC into r12 and branches to it
  – Indirect call via Global EP address in r12
Advantages and disadvantages

• Pro
  – No more function descriptors!
  – No performance regression (in fact, ~1% improvement)
  – Optimization opportunities
    • If function does not need TOC, local EP == global EP
    • Short-cut to local EP as soon as call known to be local
    • Optimize TOC save/restore just as with old ABI

• Con
  – Need to special-case dual entry points in some places
    • Linux kernel function patching
    • Valgrind transparent call redirection
    • But: in most places dual EPs “just work” transparently
In-depth: Passing parameters
Register usage

• **Goal: Pass each data type in “natural” register**
  – Integer parameters ⇒ general purpose registers
  – Floating point parameters ⇒ floating point registers
  – Vector parameters ⇒ vector registers

• **Goal: Reduce abstraction penalty**
  – OO languages wrap basic data types in a class
  – Old Power ABI passes most structs via GPRs
  – And returns most struct results in memory

• **Solution: homogeneous float/vector aggregates**
  – Classes with up to 8 aggregate elements passed in natural registers – modeled after ARM
Function return values

• Function results in same location as first input parameter
  – Homogenous float and vector aggregates in float and vector registers
  – Cap on number of registers used for GPR results (64 bytes)

• Other aggregates, unions, and arrays returned by reference in memory
  – Location provided by caller as anonymous first parameter (no change from today)
Parameter passing and variadic arguments

• Options to implement va_list in prior ABIs
  – All parameters in memory: va_list is simple pointer
  – va_list is data structure tracking registers+memory
  – va_start reconstructs linear in-memory argument list
    • Need to leave free space before on-stack params
    • Skip GPRs for parameters in FPRs or VRs
      – Allows “safe mode” for functions without prototypes by replicating FPR/VR params in GPR/memory

• ELFv2 changes
  – Eliminate parameter save area for functions that are known non-vararg and have no on-stack params
  – Preserves ABI properties, but saves stack space for *most* function calls
Stack frame reduction

- Helps in constrained environments
  - E.g., Linux kernel (limited kernel stack space)
  - Hypervisor and firmware code
- Avoid register save area in most cases
- Eliminate unused fields
  - Compiler reserved slot, linker reserved slot, VRSAVE
- Minimum stack frame size now 32 bytes
  - Old ABI required 112 bytes
Implementation Status
ELFv2 ABI Implementation Status

• Core GNU Toolchain support complete
  – Binutils, GCC, glibc, GDB

• Several packages requiring smaller changes
  – libffi, Mozilla xptcall, python-greenlet, ...
  – kernel module loader, grub2 loader, ...

• Major packages with work still in progress
  – LLVM, valgrind, mono

• Distribution status
  – Experimental porting efforts under way
    • Debian, Ubuntu, openSUSE, Fedora
    • 10000s of packages successfully built
ABI Implementation in LLVM/Clang
ELFv2 ABI implementation in LLVM/Clang

• **Current status**
  – Function call / TOC setup changes implemented
    • Patches not yet posted upstream
  – Stack frame layout changes mostly implemented
  – Homogeneous structs not yet implemented

• **Issues**
  – Code refactoring to support both ELF ABIs (and Darwin, and 32-bit SVR4)
  – Split between LLVM and Clang implementation (see example on following slides)
Function call example – source

```c
typedef struct { int a; int b; } two_ints;
typedef struct { float a; } one_float;
typedef struct { float a; float b; } two_floats;
typedef struct { long a; long b; long c; long d; } four_longs;

int a; one_float b; two_ints c; two_floats d; four_longs e; int f;

void callee (int a, one_float b, two_ints c, two_floats d,
            four_longs e, int f);

void caller (void)
{
        callee (a, b, c, d, e, f);
}
```

Stack layout at entry to callee

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BC</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>CR</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>LR</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>(-)</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>(-)</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>TOC</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>r3</td>
<td>(a)</td>
</tr>
<tr>
<td>56</td>
<td>f1</td>
<td>(b)</td>
</tr>
<tr>
<td>64</td>
<td>r5</td>
<td>(c)</td>
</tr>
<tr>
<td>72</td>
<td>r6</td>
<td>(d)</td>
</tr>
<tr>
<td>80</td>
<td>r7</td>
<td>(e.a)</td>
</tr>
<tr>
<td>88</td>
<td>r8</td>
<td>(e.b)</td>
</tr>
<tr>
<td>96</td>
<td>r8</td>
<td>(e.c)</td>
</tr>
<tr>
<td>104</td>
<td>r10</td>
<td>(e.d)</td>
</tr>
<tr>
<td>112</td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

Old ABI
Function call example – GCC asm

lwa 3,0(10)     # r10: &a
lfs 1,0(9)       # r9: &b
ld 5,0(8)        # r8: &c
ld 6,0(7)        # r7: &d
ld 7,0(11)       # r11: &e
ld 8,8(11)
ld 9,16(11)
ld 10,24(11)
lwa 0,0(4)       # r4: &f
std 0,112(1)
bl callee
nop

Old ABI
Function call example – LLVM IR

%struct.one_float = type { float }
%struct.two_ints = type { i32, i32 }
%struct.two_floats = type { float, float }
%struct.four_longs = type { i64, i64, i64, i64 }

define void @caller() {
  entry:
  %0 = load i32* @a, align 4
  %1 = load i32* @f, align 4
  %2 = load float* getelementptr inbounds (%struct.one_float* @b, i64 0, i32 0), align 4
  tail call void @callee(i32 signext %0, float inreg %2, %struct.two_ints* byval @c, %struct.two_floats* byval @d, %struct.four_longs* byval @e, i32 signext %1)
  ret void
}
### Function call example – LLVM asm

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld 12, 0(6)</td>
<td>lwa 3, 0(3)</td>
</tr>
<tr>
<td>std 12, 64(1)</td>
<td>lwa 4, 0(4)</td>
</tr>
<tr>
<td>ld 12, 0(7)</td>
<td>lfs 1, 0(8)</td>
</tr>
<tr>
<td>std 12, 72(1)</td>
<td>ld 10, 24(11)</td>
</tr>
<tr>
<td>ld 8, 24(11)</td>
<td>ld 9, 16(11)</td>
</tr>
<tr>
<td>ld 9, 16(11)</td>
<td>ld 8, 8(11)</td>
</tr>
<tr>
<td>ld 10, 8(11)</td>
<td>ld 7, 0(11)</td>
</tr>
<tr>
<td>ld 11, 0(11)</td>
<td>ld 6, 0(6)</td>
</tr>
<tr>
<td>std 8, 104(1)</td>
<td>ld 5, 0(5)</td>
</tr>
<tr>
<td>std 9, 96(1)</td>
<td>std 4, 112(1)</td>
</tr>
<tr>
<td>std 10, 88(1)</td>
<td>bl callee</td>
</tr>
<tr>
<td>std 11, 80(1)</td>
<td>nop</td>
</tr>
</tbody>
</table>

---

**Old ABI**

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>0</td>
</tr>
<tr>
<td>CR</td>
<td>8</td>
</tr>
<tr>
<td>LR</td>
<td>16</td>
</tr>
<tr>
<td>(-)</td>
<td>24</td>
</tr>
<tr>
<td>(-)</td>
<td>32</td>
</tr>
<tr>
<td>TOC</td>
<td>40</td>
</tr>
<tr>
<td>r3</td>
<td>48</td>
</tr>
<tr>
<td>r3</td>
<td>56</td>
</tr>
<tr>
<td>r3</td>
<td>64</td>
</tr>
<tr>
<td>r3</td>
<td>72</td>
</tr>
<tr>
<td>r7</td>
<td>80</td>
</tr>
<tr>
<td>r7</td>
<td>88</td>
</tr>
<tr>
<td>r7</td>
<td>96</td>
</tr>
<tr>
<td>r10</td>
<td>104</td>
</tr>
<tr>
<td>f</td>
<td>112</td>
</tr>
</tbody>
</table>

© 2014 IBM Corporation
ABI implementation: LLVM vs. Clang

• Problems to be solved
  – Do not use “byval” for anything completely in registers
    • In ELFv2, if everything is in register, there is no parameter save area, so we cannot “stage” there
    • In any case, staging all structs is inefficient
  – Detect homogeneous structs in Clang and/or LLVM ?
    • Note: “float” struct member uses 4 bytes of stack; stand-alone “float” variable uses 8 bytes of stack!
  – Do I need to track registers in Clang?
    • To know for sure whether argument will end up in regs
    • Currently done for x86_64 target
Summary
Summary

- New little-endian 64-bit PowerPC architecture
- Opportunity to implement new ABI
  - Largely aligned with old PowerPC64 ABI, but ...
  - No more function descriptors
  - Improved parameter passing
- Implementation status
  - Several Linux distributions in experimental porting
  - Core GNU toolchain fully implemented
  - Clang/LLVM implementation in progress
    - Still need to resolve some issues
Questions