Porting to 64-bit GNU/Linux Systems

Andreas Jaeger
SuSE Linux AG
aj@suse.de, http://www.suse.de/~aj

Abstract

More and more 64-bit systems are showing up on the market—and developers are porting their applications to these systems. Most code runs directly without problems—but there is a number of sometimes quite subtile problems that developers have to be aware of for portable programming and porting.

This paper illustrates some problems on porting an application to 64-bit and also shows how use a 64-bit system as development platform for both 32-bit and 64-bit code. It will give hints especially to application and library developers on writing portable code using the GNU Compiler Collection.

1 Introduction

With the introduction of AMD's 64-bit architecture, AMD64, implemented in the AMD Opteron and Athlon64 CPUs, another 64-bit processor family enters the market and users are going to buy and deploy these systems. A new architecture offers new challenges for both system developers (compare [JH]) and application developers.

This paper will give hints especially to application and library developers to write portable code and make use of their 64-bit development machine. While the paper discusses general 64-bit and porting problems specific to other platforms, the AMD64 platform is used as primary example. Other architectures that the author has access to and is familiar with are discussed also. A brief characteristic of these 64-bit Linux platforms[1] is given in table 1.

Differences between platforms and therefore the need to port software can be attributed to at least one of:

Compiler Different compilers have different behavior. This can mostly be avoided with using the same version of the GNU compilers.

Application Binary Interfaces (ABI) An ABI specifies sizes of fundamental types, function calling sequence and the object format. In general the ABI is hidden from the developer by the compiler.

CPU The effect of different CPUs is mainly visible through the ABI. The differences visible to developers include little or big endian, whether the stack grows up or down, or whether the fundamental size is 32-bit or 64-bit.

C Library Different C libraries might not implement the same subset of functions or

[1] The only missing 64-bit platforms that I am aware of are MMIX and SuperH SH 5 but there is no Linux port for them.

have architecture dependent versions. The GNU C Library tries to unify this but there are always architecture dependent differences.

Kernel All access to the Linux kernel is done through functions of the C Library. A newer kernel might have additional functionality that the C Library then can provide.

Application developers will mainly have portability problems due to different CPUs and different ABIs and the discussion here will concentrate on these.

The paper is structured as follows: Section 2 mentions why 64-bit programs are advantageous. The following section discusses execution of both 32-bit and 64-bit programs on one system and development on such a system. Section 4 shows how easy porting should be and then goes into all the subtleties and problems that nevertheless arise.

2 Advantages of 64-bit Programs

The main limitation of 32-bit programs that push developers to 64-bit programs is the limited address space. A 32-bit program can only address 4 GB of memory. Under a 32-bit x86 kernel the available address space is at most 2-3 GB (3.5 GB with a special kernel and static linking of an application) since the kernel also needs some of that memory. Nowadays applications need larger and larger address spaces and performance can be greatly improved with large caches which is a benefit especially for databases.

Besides larger address space most recent 64-bit processors introduce additional features over the previous processor generation for improved performance.

As an example the 64-bit AMD Opteron processor has some architectural improvements, like a memory controller integrated into the processor for faster memory access which eliminates high latency memory structure. Programs written in 64-bit mode for AMD Opteron take implicitly advantage of this but also of further new features:

- 8 additional general purpose and 8 additional floating point registers
- RIP addressing (instruction-pointer relative addressing mode) to speed up especially handling of shared libraries[JH].
- A modern Application Binary Interface [AMD64-PSABI].
- A large address space (currently 512 TB per process).

3 64-bit and 32-bit Programs on One System

The CPU architects of the 64-bit architectures AMD64, MIPS64, Sparc64, zSeries and PowerPC64 designed their CPUs in such a way that these 64-bit CPUs can execute 32-bit code natively without any performance penalty. The most sold 64-bit platform is the MIPS architecture but it—due to its usage nowadays mainly in embedded systems—mainly runs in 32-bit mode. Under Linux the 64-bit platforms PowerPC64 and Sparc64 in general only use a 64-bit kernel but have no significant 64-bit application base.

All these architectures nevertheless share the way that their 32-bit support is done. The support of two architectures is commonly called

Architecture	`uname -m`	Size	Endian	Libpath	Miscellaneous
Alpha	`alpha`	LP64	little	lib	
AMD64	`x86_64`	LP64	little	lib64	executes x86 code natively
IPF	`ia64`	LP64	little	lib	executes x86 code via emulation
MIPS64	`mips64`	LP64	both	lib64	executes MIPS code natively
PowerPC64	`ppc64`	LP64	big	lib64	executes PowerPC code natively
Sparc64	`sparc64`	LP64	big	lib64	executes Sparc code natively
PA-RISC64	`parisc64`	LP64	big	—	only kernel support, no 64-bit user land, executes 32-bit PA-RISC code natively
zSeries (s390x)	`s390x`	LP64	big	lib64	executes s390 code natively

Table 1: 64-bit Linux Platforms

"biarch support" and there's also the general concept of "multi-arch support."

A 64-bit architecture that can execute 32-bit applications natively offers some extra challenges for developers:

- The kernel has to support execution of both 32-bit and 64-bit programs.
- The system has to be installed in such a way that 32-bit and 64-bit libraries of the same name can exist on one system.
- The tool chain should handle development of both 32-bit and 64-bit programs.

3.1 Kernel Implications

The kernel side is not part of this paper but the requirements for the kernel implementation should be stated:

- Starting of programs for every architecture supported by the ABI, e.g. for both 32-bit and 64-bit.
- System calls for every architecture in a way that is compatible to the corresponding 32-bit platform. For example a program that runs on x86 should run on AMD64 without any changes.

  One problem here is the `ioctl()` system call which allows to pass any kind of data to the kernel including complex data structures. Since the kernel needs to translate these data structures to the same structure for all supported architectures, some `ioctl()`s might only be supported for the primary architecture. This restriction only hits administration programs, like LVM tools.

3.2 Libraries: `lib` and `lib64`

If a system only supports execution of one architecture, all libraries will be installed in paths ending with `/lib` like `/usr/lib` and user-level binaries in paths ending with `/bin`, e.g. `/usr/bin`. But if there's more than one architecture to support, libraries will exist in flavors for each architecture but with the same name, e.g. there's a `libc.so.6` for 32-bit x86 and one for 64-bit code on an AMD64 system. The problem now is where to install these libraries.

Following the example set by the Sparc developers, all the other 64-bit biarch platforms install the 64-bit libraries into paths ending with `/lib64`, e.g. `/usr/X11R6/lib64`. The 64-bit dynamic linker is configured to search

these library paths. For 32-bit libraries nothing has been changed.

This setup has the advantage that packages build for the 32-bit platform can be installed without any change at all. For them everything is the same as on the corresponding 32-bit platform, no paths are changed at all. For example the binary x86 RPM package of the Acrobat Reader can be installed directly on AMD64 systems and works without any change at all.

For 64-bit programs a little bit more work is needed since often configure scripts search directly the library paths for certain libraries but then find only the 32-bit library in e.g. `/usr/lib` or makefiles have paths hard-coded. Configure scripts created by GNU `autoconf` offer an option to specify the library install path directly and if you use RPM, you can use for example the following in your spec file:

```
configure --prefix=/usr --libdir=%{_libdir}
```

Also `ldconfig` handles both 32-bit and 64-bit libraries in its configuration (`/etc/ld.so.conf`) and cache files (`/etc/ld.so.cache`). `ldconfig` marks 64-bit libraries in the cache so that the dynamic linker can easily detect 32-bit and 64-bit libraries.

3.3 Development for Different ABIs

GCC can be build as a compiler that supports different ABIs on one platform. Depending on the architecture a number of different ABIs or instruction sets are supported, e.g. for ARM it is possible to generate both ARM and Thumb code. The GNU binutils also support these different ABIs.

The framework is especially useful for a biarch compiler and the 64-bit GNU/Linux platforms AMD64, MIPS, Sparc64 and zSeries (s390x) have support to generate code not only for the 64-bit ABI but also for the corresponding 32-bit (31-bit for zSeries) ABI. The PowerPC64 developers have not yet implemented this in GCC but I expect that they follow the same road.

Note that in the following text only the C compiler (`gcc`) is mentioned. The whole discussion and options are also valid for the other compilers in the GNU Compiler Collection: The C++ compiler (`g++`), the Ada compiler (`gnat`), the Fortran77 compiler (`g77`) and the Java compiler (`gcj`).

3.3.1 The AMD64, Sparc64 and zSeries Way

For AMD64, Sparc64 and zSeries the compiler generates by default 64-bit code. To generate 32-bit code for x86 (on AMD64) or for Sparc (on Sparc64), the compiler switch `-m32` has to be given to GCC. Compilation for 31-bit zSeries on a 64-bit zSeries needs the `-m31` option. The assembler and linker have similar switches that GCC passes to them. The compiler also knows about the default library paths, e.g. `/usr/lib` vs. `/usr/lib64` and invokes the linker with the right options. An example compile session is given in figure 1.

3.3.2 MIPS and its ABIs

MIPS does not only support support 32-bit and 64-bit programs, it also support two different ABIs for 32-bit programs. The three ABIs can be summarized as follows:

Name	Library Path	GCC Switch
o32 (old 32-bit)	`/lib`	`-mabi=o32`
n32 (new 32-bit)	`/lib32`	`-mabi=n32`
n64 (64-bit)	`/lib64`	`-mabi=64`

```
$ gcc hello.c -o hello64
$ gcc -m32 hello.c -o hello32
$ file ./hello32 ./hello64
./hello32: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
./hello64: ELF 64-bit LSB executable, AMD x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
$ ldd ./hello32 ./hello64
./hello32:
        libc.so.6 => /lib/libc.so.6 (0x40029000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
./hello64:
        libc.so.6 => /lib64/libc.so.6 (0x0000002a9566b000)
        /lib64/ld-linux-x86-64.so.2 =>
                        /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)

$ gcc -L /usr/X11R6/lib -L /usr/X11R6/lib64 xhello.c -o xhello64 -lX11
/usr/lib64/gcc-lib/x86_64-suse-linux/3.3/../../../../x86_64-suse-linux/bin/ld:
skipping incompatible /usr/X11R6/lib/libX11.so when searching for -lX11
$ gcc -m32 -L /usr/X11R6/lib -L /usr/X11R6/lib64 xhello.c -o xhello32 -lX11
$ ldd ./xhello64 ./xhello32
./xhello64:
        libX11.so.6 => /usr/X11R6/lib64/libX11.so.6 (0x0000002a9566b000)
        libc.so.6 => /lib64/libc.so.6 (0x0000002a95852000)
        libdl.so.2 => /lib64/libdl.so.2 (0x0000002a95a94000)
        /lib64/ld-linux-x86-64.so.2 =>
                         /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)
./xhello32:
        libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x40029000)
        libc.so.6 => /lib/libc.so.6 (0x400f8000)
        libdl.so.2 => /lib/libdl.so.2 (0x4022e000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Figure 1: Example Compile Sessions on AMD64

Note that the Linux Kernel so far supports only the o32 ABI completely, support for the other two is currently been worked on.

3.3.3 Toolchain

GCC knows how to invoke assembler and linker to generate 64-bit or 32-bit code. Therefore in general GCC should be just passed the right option for compilation and linking. In cases where developers really need to call the binary utilities[2] directly for 32-bit code, there's a short list of these options for the GNU binutils in table 2. The user can inquiry most of these options directly with calling `gcc -v` to print out the commands issued by the compiler.

[2]Calling these directly might also harm since GCC passes extra options to the binary utilities. For example the linker `ld` will not produce correct C++ binaries if not called with the right set of options which GCC does automatically.

Tool	Option for 32-bit code		
	AMD64	Sparc64	zSeries
ar	No option needed		
as	`--32`	`-32`	`-m31`
gcc,g++,...	`-m32`	`-m32`	`-m31`
ld	`-m elf_i386`	`-m elf32_sparc`	`-m elf_s390`
nm	No option needed		
strip	No option needed		

Table 2: Options for 32-bit Code Generation on 64-bit Architectures

3.3.4 Caveat: Include Files for Multi-Arch Compilation

The support for different ABIs on one systems has one problem: What happens if different versions of a library are installed that have a different interface? For example, the 64-bit library could be an older version than the 32-bit library and the newer version has changed data types or signatures of functions. Since there is only one include directory for all ABIs (there is no `/usr/include64`!), the system administrator has to take care that installed header files are correct for all ABIs and libraries. In the worst case the include file has to include support for each ABI using preprocessor conditionals. As an example, the GNU C Library has quite a few kernel dependent interfaces that are different between architectures. The include files for e.g. AMD64 therefore have—where necessary—constructs like the following (from <`bits/fcntl.h`>):

```
#include<bits/wordsize.h>
[...]
# if __WORDSIZE == 64
#  define O_LARGEFILE 0
# else
#  define O_LARGEFILE 0100000
# endif
```

3.3.5 Debugging

The GNU Debugger (gdb) is currently getting enhanced to be able to debug a number of different architectures and ABIs. So, in the future, we could have a GDB that debugs all binaries that can run on one architecture, e.g. both 32-bit x86 and 64-bit programs on AMD64 systems. Currently this is not possible and therefore a separate debugger has to be used for every ABI. For example, SuSE Linux on AMD64, has a `gdb` binary to debug AMD64 programs and a `gdb32` binary for x86 programs.

The system tracer `strace` has on some architectures, e.g. AMD64 and Sparc64 already the capability to trace both 32-bit and 64-bit programs. On other systems both a 32-bit and a 64-bit version needs to be put in place with different names.

3.3.6 Changing the Personality

The output of `uname -m` is used by e.g. `configure` to check for which architecture to build. This can cause problems if you build on a 64-bit system for the corresponding 32-bit architecture since then `configure` might decide that this is a cross-compilation instead of a native compilation. For such cases the output of `uname -m`, the so called personality,

can be changed with a special system call. The personality is inherited by children from their parents. There exists a user space program to change the personality and it can be used e.g. on AMD64 as:

```
$ uname -m
x86_64
$ linux32 bash
$ uname -m
i686
```

to create a shell with changed personality for further development.

The name of the user space program is different on different architectures, the following list contains those names that we are aware of:

Architecture	Personality Tool
AMD64	`linux32`
PowerPC64	`powerpc32`
Sparc64	`sparc32`
zSeries	`s390`

3.4 Development

So, with the complete toolchain supporting different ABIs, it is now possible to develop both 64-bit and 32-bit programs on one machine. Instead of having two machines heating the room, a developer can use only a 64-bit box as development machine and still produce and test 32-bit code.

To develop 32-bit code on an AMD64 system, the developer has to add the `-m32` option to the compiler flags, no other changes are needed in general.

For the development of native 64-bit AMD64 code on the same machine, the only change might be to change the library path if another library path as `/usr/lib64` is used. It is even safe to give both the 32-bit and the 64-bit path, the linker will find the right library directly (but emit warnings) as shown in figure 1.

4 64-bit Porting: Hints and Pitfalls

Porting to a 64-bit system is not a problem for portable programs. Unfortunately most programs are not really portable and therefore need to be changed to run correctly on another platform.

The porting effort on GNU/Linux platforms is lower than e.g. between Unix and GNU/Linux since all GNU/Linux platforms use the GNU C Library. The C Library tries to use a common implementation and headers for all platforms which eases portability. Using the same C Library cross platforms means:

- Usage of the same functions: The set of functions is the same in general. Only a few functions are architecture specific and those are needed in general to access hardware which is platform specific.
- A different layout of structures: The C Library implements the different processor specific ABIs and therefore structures can have different length and members.

Therefore a program that is written portable, without reference to platform specific features, in general can be easily ported from on platform to the other, e.g. from 32-bit to 64-bit.

Each platform has its own special “features,” meaning that some non-portable code works on all platforms except one. Keeping these problems in mind helps writing portable code and eases debugging of non-portable code.

Most of the problems arise in C and therefore this language is used everywhere in this paper. Some of these problems might not arise in C++ since C++ has some stricter rules.

The general problem is that sizes of fundamental types on different platforms, and especially between 32-bit and 64-bit platforms, are different and therefore not all types are interchangeable.

4.1 "Portable" x86/AMD64 Inline Assembler

There are some things that can not be done portably in general. One issue is inline assembler. For processors from the same family, like x86 and AMD64 processors, often assembler code can be shared. But this is not possible between different architectures.

A small example for inline assembler on x86 and AMD64 is the following function:

```
/* ffs -- find first set bit in a
   word, counted from least
   significant end.  */
int
__ffs (int x)
{
  int cnt, tmp;
  /* Count low bits in X; store in
     %1.*/
  asm ("bsfl %2,%0\n"
       "cmovel %1,%0\n"
   /* If number was zero, return
      -1.*/
   : "=&r" (cnt), "=r" (tmp)
   : "rm" (x), "1" (-1));
  return cnt + 1;
}
```

This would be compiled by GCC for x86 to:

```
mov     $0xffffffff,%eax
mov     %eax,%edx
bsf     0x4(%esp),%ecx
cmove   %edx,%ecx
mov     %ecx,%eax
inc     %eax
ret
```

The assembler for AMD64 looks like this:

```
mov     $0xffffffff,%eax
mov     %eax,%edx
bsf     %edi,%ecx
cmove   %edx,%ecx
mov     %ecx,%eax
inc     %eax
ret
```

This example worked fine since `int` is 32-bit on both x86 and AMD64 and the same instructions can be used. For datatypes `long` this scheme cannot be used since it's 32-bit on x86 and 64-bit on AMD64. The size of `long long` is 64-bit on both architectures but since AMD64 has 64-bit registers code can be written that is more efficient.

Using the inline assembler in that function made it possible for the developer to ignore the different passing conventions in this example. For x86 the parameter `x` is passed on the stack (`0x4(%esp)`) and for AMD64 in the lower 32 bits of register `RDI` (`%edi`).

4.2 Sizes and Alignment of Fundamental Datatypes and Structure Layout

On 64-bit platforms pointers and the type `long` have a size of 64 bits while the type `int` uses 32 bits. This scheme is known as the LP64 model and is used by all 64-bit UNIX ports. A 32-bit platform uses the so-called ILP32 model: `int`, `long` and pointers are 32 bits.

The differences in sizes (in bytes) between the 32-bit x86 and the 64-bit AMD64 are summarized in the following table:

Type	i386	AMD64
`long`	4	8
pointer	4	8
`long double`	12	16

Besides the different sizes of fundamental types, different ABIs specify also different alignments. A `double` variable, for example, is aligned on x86 to 4 bytes but aligned to 8 bytes on AMD64 despite having the same size of 8 bytes. Structures will therefore have a different layout on different platforms. Additionally some members of structures might be in a different order or the newer architecture has additional members that could not have been added to the older one.

It is therefore important not to hard code any sizes and offsets. Instead the C operator `sizeof` has to be used to inquire sizes of both fundamental and complex types. The macro `offsetof` is available to get the offsets of structure members from the beginning of the structure.

4.2.1 `int` vs. `long`

Since the sizes of `int` and `long` are the same on a 32-bit platforms, programmers have often been lazy and used `int` and `long` interchangeably. But this will not work anymore with 64-bit systems where `long` has a larger size than `int`.

A few examples:

- Due to its size a pointer does not fit into a variable of type `int`. It fits on Unix into a `long` variable but the `intptr_t` type from ISO C99 is the better choice.
- Untyped integral constants are of type (unsigned) `int`. This might lead to unexpected truncation, e.g. in the following snippet of non-portable code:

```
long t = 1 << a;
```

  On both a 32-bit and a 64-bit system the maximal value for a can be 31, since the type of `1<<a` is `int`. To get a shift done in 64-bit (a `long` calculation), `1L` has to be used.

- The type of identifiers of an enumeration is implementation defined but all constants get the same type. GCC by default gives them type `int`, unless any of the enumeration constants needs a larger type.

4.3 Function Prototypes

If a function is called in C without function prototypes, the return value is `int`—and that's a 32-bit type on all 64-bit Linux platforms. For arguments the integer promotions are performed and arguments of type `float` are promoted to `double`.

Such a missing prototype can easily lead to a segmentation fault. For example if `malloc()` or `memcpy()` are used without a prototype, the resulting binary might break because of:

`malloc()` The return value is a 32-bit entity and therefore only half of the bits of the returned address might be stored in the variable that holds the return value making the pointer invalid.

`memcpy()` The first two arguments are pointers that take the source and target address. If, instead of the 64-bit pointers, only the lower 32 bits are passed to `memcpy()`, the function will access random memory (note this can only happen if the pointer has been assigned to a variable of `int` and that variable is used for passing).

4.4 Variable Argument Lists

The problem with variable argument lists is the same problem as with missing function prototypes: At the call side an argument is passed to a function but the function expects an argument of a different size.

If you pass in a 32-bit value, it is normally passed in 64-bit registers or on the stack as 64-bit value. The question now is what to do with the unused 32 bits? The 32-bit value can be zero-extended so that the unused bits are all zero, it can be sign-extended giving all zeros or all ones, and it can be left unspecified (as on AMD64). If the called function expects now a 64-bit value where it gets a 32-bit value, the function might not work as expected.

The important rules are:

- If you pass 32-bit values, like variables of type `int`, the called function has to take out 32-bit values.
- If the function expects 64-bit values, like `long` or pointers, the caller has to pass 64-bit values. Note that `0` is not the same as a `NULL` pointer since those have different sizes.

Another topic is usage of `va_lists`. You cannot copy variables of this type directly. This works on those platforms that use a pointer to implement `va_lists` but not on others. Use instead the function-like macro `va_copy`.

4.5 Function Pointers

Often programmers assume that all pointers have the same format but this is not guaranteed by the ISO C standard.

On IPF, PA-RISC and PowerPC64 a pointer to a function and a pointer to an object are represented differently. For example on IPF, a function pointer points to a descriptor containing the function address and the value of the GP (global pointer, used with shared libraries) register:

```
struct ia64_fdesc {
  uint64_t func;
  uint64_t gp;
  };
```

The GP register needs to be set with the right value before calling any function.

This means the following should not be done in a portable program:

Compare function pointers Since there can be more than one descriptor for any function, different function pointers for the same function will have different values.

Locate function The function pointer will not point directly to the function, so it cannot be used easily to find the actual code of the function.

Construct function pointer from data address This will fail since the GP register will not be setup correctly.

4.6 Using Bitwidth-Dependent Types Portably

Some applications depend on specific sizes for their datatypes. As has been mentioned before, this cannot be done portably in general. ISO C99 introduced a new header file `stdint.h` that defines datatypes having specified widths and a corresponding set of macros. The following types are also specified:

Exact-width integer types Signed integer types of the form `intN_t` (unsigned: `uintN_t`) with width N are defined in general with widths 8, 16, 32, or 64. A `int32_t` is therefore a signed 32-bit integer.

Minimum-width integer types The types `int_leastN_t` for signed and `uint_leastN_t` for unsigned integers with a width of at least N bits are defined. The widths 8, 16, 32 and 64 are required to be supported.

Fastest Minimum-width integer types The types `int_fastN_t` for signed and `uint_fastN_t` for unsigned integers with width at least N bits are defined as types that are usually the fastest of all integer types having at least this width. Width of 8, 16, 32 and 64 are required to be supported.

Integer types holding pointers The integer types `intptr_t` and `uintptr_t` can hold a pointer, a conversion between pointer and this integer type is always possible.

Greatest-width integer types The integer types `intmax_t` and `uintmax_t` hold any value of any signed/unsigned integer type.

Note that an ISO C99 implementation does not need to implement all of these types. The GNU C Library implements all of them for all platforms.

In addition to these types a number of macros are defined to give the limits of the types.

Inclusion of the header `inttypes.h` defines additional macros for format specifiers both for `printf` and `scanf` for these types, and some conversion functions like `strtoimax`.

An example of the usage of the types and the format specifier for printing is:

```
#include <inttypes.h>
#include <stdio.h>
int
main (void) {
  intmax_t u = INTMAX_MAX;
  printf("The largest signed integer"
         " is: %" PRIdMAX "\n", u);
  return 0;
}
```

4.7 Using `printf` and `scanf`

ISO C99 introduced a few new format specifiers to allow printing and scanning of certain types that might have architecture dependent size. These are `%p` for printing a pointer value and the `%Z` size modifier for arguments of type `size_t`. An example:

```
...
void *p;
printf("p has value %p and "
       "size %Zd\n", p, sizeof(p));
```

4.8 Unsigned and Signed Chars

The ISO C Standard does not define the signedness of the type `char`.[3] A definition like `char foo;` creates an unsigned variable on some platforms but a signed one on others. If you use variables of type `char` as small integers, you should specify whether you need a signed or an unsigned type. Also comparisons with `char` variables should take this into account, the following code snippet will not give the desired outcome if `char` is unsigned:

```
char c;
if (c < 0)
  puts("Non-ascii character");
```

[3] Note that this is not a 64-bit problem but it is one of those differences you'll notice when porting and is therefore worth mentioning.

During compilation GCC should generate the warning "`warning: comparison is always false due to limited range of data type`".

Platforms with an unsigned char type are both 32-bit and 64-bit versions of S390 and PowerPC. GCC has the options `-fsigned-char` and `-funsigned-char` to change the signedness of type `char`.

4.9 Evaluation of Floating-Point Arithmetic

A common confusion happens when suddenly algorithms using floating-point arithmetic give different results. The IEEE754 standard defines that the basic operations have to be exact. But nevertheless, results might vary between architectures.

The problem happens with operations of type `float` and `double` since on the popular x86 architecture these operations are evaluated in the x87 FPU in `long double` precision. The compiler might choose to leave intermediate results (with a type of `long double`) in the x87 FPU or convert them back to the target type. Depending when this conversion happens, different rounding errors occur.

A small example to show the differences is:

```
#include <stdio.h>
int
main (void)
{
  float b, c;

  b = 1 / 3.0f;
  c = b * 3.0f - 1.0f;
  printf ("c: %.20f\n", c);
  return 0;
}
```

Compiling and executing this program on an Linux/AMD64 system gives different results between 32-bit x86 and 64-bit binaries:

```
$ gcc t.c -m32 -o t32
$ gcc t.c -o t64
$ ./t32
c: 0.00000002980232238770
$ ./t64
c: 0.00000000000000000000
```

Note that the example gives the same results if compiled with optimization since without optimization `b` is stored in memory as type `float` but with optimization `b` is left in the FPU.

ISO C99 defines the macro `FLT_EVAL_METHOD` for this in the header <`float.h`>. It is set to:

0 If evaluation is done with the range and precision of the type. This is the value on nearly all Linux systems.

1 If evaluation of expressions of type `float` and `double` is done to the range and precision of `double` and of `long double` to the range and precision of `long double`.

2 If all evaluations is done to the range and precision of type `long double`. This is the value on Linux/x86.

-1 Indeterminable.

This problem with different results due to the evaluation of floating-point arithmetic is not a genuine 64-bit problem but a problem between x86 code and all other platforms and therefore might hit developers porting from x86 to other platforms, e.g. to AMD64.

4.10 Shared Libraries

Most architectures have the constraint that shared libraries need to be compiled as PIC-code using the `-fPIC` switch to GCC. Even for those architectures that allow it, like x86, it is not desirable to do so since a shared library should live once in the memory and get then shared by all applications using it. But non-PIC code cannot be shared.

Architectures that force to use `-fPIC` for shared libraries include AMD64, IPF, and PA-RISC.

4.11 How to Check for 64-bit?

Starting with GCC 3.4, **all** LP64 platforms will define the macros `__LP64__` and `_LP64` that can be used e.g. in preprocessor defines. Earlier GCC releases define this macro only on a few platforms or OSes. For GCC 3.2 and 3.3, the macros are defined on NetBSD, for IPF (every OS), for PA-RISC (every OS) and for AMD64 running Linux (starting with GCC 3.2.3).

In general it is possible to check for 64-bit with the architecture builtins of GCC, e.g. with:

```
#if defined(__alpha__)\
 ||defined(__ia64__)\
 ||defined(__ppc64__)\
 ||defined(__s390x__)\
 ||defined(__x86_64__)
```

but this needs to be enhanced for each new 64-bit platform. The better solution is to write portable code that does not need to check for architecture details.

4.12 Optimized Functions, Macros, and Builtins

The GNU Compiler Collection uses the same optimizations on all platforms but some of them are tuned in different ways and others need help from the architecture specific backend. One area were this occurs especially are builtin functions.

A function like `strlen` can be implemented in the following ways:

As builtin in GCC The compiler can detect that e.g. the arguments to `strlen` are constant and evaluate the function at compile time. It can also optimize the function to an inline function and do a loop instead of calling the external `strlen` function. This can be disabled with `-fno-builtin` or `-fno-builtin-`*`function`*.

As macro in Glibc The C Library implements a number of functions as macros. The string inline functions can be disabled with a definition of `__NO_STRING_INLINES`, some of them are only enabled if `__USE_STRING_INLINES` is passed. For details check the header `/usr/include/string.h` directly. Inlining of mathematical functions can be disabled by defining `__NO_MATH_INLINES`. Also it is allowed to disable a specific macro like `#undef strlen`.

As function in Glibc ISO C99 forces to implement all required functions as functions. Therefore for example `strlen` will always be in the C Library.

Some developers decide to override the C Library functions and write their own optimized implementation. This works fine for one system consisting of a specific CPU, a specific C

Library and GCC version. But going to another architecture, better optimizations might be possible, e.g. reading 8 bytes at once instead of 4 in `strlen`, or current code is penalized, e.g. alignment is mandatory for 8 byte access.

So, instead of writing something just for one program, it should be done in a generic way in GCC or glibc so that all programs can benefit from one optimization.

A number of functions in Glibc are written in hand-optimized assembler for some architectures and where this is not done, a good C implementation is used. On AMD64 the compiler has builtins for the common string functions and also for some mathematical functions and uses them depending on the arguments and enabled compiler optimizations, e.g. `-Os` disabled most builtins since they would increase size.

The pitfalls regarding porting here are that a programs does optimizations that are not valid for a new architecture or does not expect that a function might be implemented as a macro or builtin.

4.13 Useful Compiler Flags

An incomplete list of GCC compiler flags that might be useful for porting code:

-Wall Enables a number of default warnings, should be used for all code

-W Enables additional warnings. Some of them are hard to avoid so this might not be useful for all code.

-Wmissing-prototypes Warn about missing prototypes, this is especially important for 64-bit ports.

5 Conclusion

Despite the different problems we encountered at SuSE while porting to the various 64-bit platforms (first for Alpha, later for IPF, zSeries, AMD64 and PowerPC64), the number of packages with actual problems is getting smaller and smaller since code has less platform specific assumptions and is more portable.

Also development of the toolchain has been improved recently and there is more focus on creating bi-arch toolchains to allow compilation for different ABIs on one system.

I hope that the problems mentioned and explained will help further in writing portable and efficient code.

6 Acknowledgments

Porting to any new architecture means building on the foundations that others have led, learning from their experiences and tackling with others all the subtleties of non-portable programming. I'd like to thank especially my colleagues Andreas Schwab for help in lots of debugging sessions and bug fixing of tools and programs, Stefan Fent and Stefan Reinauer for driving the port of SuSE Linux to AMD64 and thereby encountering many of the problems mentioned in this paper, Jan Hubička and Michael Matz for porting and fixing GCC and the ABI on AMD64—and all of them for their discussions on these issues. Thanks also to Michael Matz and Evandro Menezes for reviewing the paper.

References

[AMD64] *AMD64 Architecture Programmer's Manual*, AMD (2003).

[Opteron] *Software Optimization Guide for the AMD Opteron™ Processor*, AMD (2003).

[AMD64-PSABI] *UNIX System V Application Binary Interface; AMD64 Architecture Processor Supplement, Draft*, (Ed. J. Hubička, A. Jaeger, M. Mitchell), `http://www.x86-64.org`, (2003)

[i386-ABI] *UNIX System V Application Binary Interface; IA-32 Architecture Processor Supplement*, Intel (2000).

[ISOC99] *Programming Languages—C*, ISO/IEC 9899:1999 (1999)

[IEEE754] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE754-1985 (1985).

[JH] *Porting GCC to the AMD64 Architecture*, Jan Hubička, GCC Summit (2003).