

A look at 64- versus 32-bit userspace performance

Olof Johansson

olof@lixom.net

January 25, 2006

Abstract

This paper will take a look at the two common Linux platforms that are able to mix 32- and 64-bit userspace programs: ppc64 and x86_64. It will investigate the performance behaviour of the applications depending on if they are compiled as 32 or 64-bit binaries, and explain why we see the differences we do. It will compare the two modes and explain why running on x86_64 as a 64-bit application likely gives a performance improvement, while it is much less likely to do so on ppc64.

1 64-bit computing

A common term used for 64-bit computing is LP64 – “longs and pointers are 64-bit”. The corresponding term for 32-bit is ILP32 – “integers, longs and pointers are 32-bit”. Both terms will be used throughout this text.

Moving from ILP32 to LP64 is by no means a silver bullet that will solve performance bottlenecks and make everyone’s life happier. Depending on the underlying architecture, there are other performance impacts to consider when deciding which way to go.

1.1 Background

While this was a hot topic of discussion a few years ago, these days few people challenge the statements that 64-bit computing have several real-life valuable benefits over 32-bit:

1. A much larger address space, allowing for example memory mapping of whole databases and leaving the need for paging parts of it in and out to the OS

2. Larger integer registers improves performance for some operations such as encryption
3. The larger address space removes some of the limitations of operating systems when it comes to dealing with more memory than can be addressed directly (i.e. highmem, 4G+4G kernel patches, etc).

1.2 AMD’s x86_64 architecture

In the late nineties, Intel were gathering all their eggs in one basket: They were going to develop a brand new 64-bit architecture and move most of their existing and future customer bases over to it.

We all know how well that went.

Luckily, AMD had a competing strategy that they brought to market: Instead of developing a brand new architecture, they expanded the existing 32-bit x86 architecture to have an expanded 64-bit execution mode. In addition to the basic changes of register and word sizes, they also took the opportunity to fix one of the biggest limitations of the x86 architecture today: They doubled the number of available general-purpose registers in the processor.

So far, the x86_64 architecture has been a success. Intel has even come around, and most of their current processors include features much like the AMD processors to enable 64-bit processing. IA64 is still alive, but most of the customer base that were main targets for it seem to have moved to the x86_64 architecture or to other architectures altogether.

x86_64 also has the benefit of being an extension to x86, and as such, it can still run old programs in native mode. IA64 has binary translators and emulators to do the same thing, but emulation and translation always brings challenges of performance impacts and slight behavioural differences that might make a subtle difference for some

workloads. Bottom line: x86_64 combines the best of the two worlds of 64-bit computing and legacy support.

1.3 IBM's PPC64 architecture

The PowerPC processor architecture was designed with a 64-bit implementation in mind. IBM (together with partners) have to date been the only maker of 64-bit PPC processors, but more are on the way.

The 64-bit extensions of PowerPC change less of the underlying architecture than AMD's changes to x86. There are already plenty of registers on PPC, so there was no reason to modify those aspects of the architecture. Most changes are instructions for loading and storing 64-bit register contents, extending from 32- to 64-bit as well as revised meanings of existing instructions where they operate on 64-bit instead of 32-bit registers.

For a userspace application, there aren't that many differences between running 64- versus 32-bit. While the 64-bit families do bring changes to how the operating system uses the hardware, most changes are needed even if the same processor is used in 32-bit mode. Most substantial changes have been to how the MMU is programmed, since it had to change quite a bit to accommodate larger addressing ranges.

When it comes to userspace, what has been the biggest difference between the two binary formats are the ABIs, which contains the rules for how an executable is laid out, how shared libraries work and are called, and all other details that are needed to make sure that software behaves the same no matter in which environment it's running in.

2 Binary formats

For example, consider a simple hello world application:

```
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
}
```

This will result in a simple binary file with just a couple of symbols linked in. The most obvious one is the func-

tion printf(), which is called from the main function.

In the 32-bit ELF format, the loader would, through information in the symbol table, know to go in and replace a placeholder instruction that calls the printf() function with one that actually will call the real function, depending on what address the standard C library is linked at and thus where in memory that function is at.

The PPC64 ELF format works quite a bit differently. Instead of the loader going in and changing the code itself, there is a table as part of the binary that contains function descriptors. Each function descriptor contains the entry point for the function in question, together with an environment variable to set before calling it (in some languages) and the value of the TOC pointer for the called function. For more details, see section 3.5.11 of the 64-bit PowerPC ELF ABI Supplement[1].

3 Binary file sizes

3.1 PPC

Looking at gcc 4.0.1's cc1 for i686 target:

64-bit:

text	data	bss	dec	hex
5020366	528576	561216	6110158	5d3bce

32-bit:

text	data	bss	dec	hex
4759812	9048	504524	5273384	507728

Major differences are in data and text:
519528 bytes (5741%) more data
260554 bytes (5.5%) more text
56692 bytes (11%) more bss

Where does the all that extra data come from?

objdump -h on the binaries tells us that .rodata is footnoteseizer on 64-bit (0x6670f instead of 0xa41e8), but that's outside of what's reported by 'size' as data anyway. Regular .data is 0x3558 vs 0x2060, which is easy to see assuming larger data types for long and pointer sizes.

Besides that, the major differences are sections that don't

exist in the 32-bit binary:

```
17 .data.rel.ro 000346d8 (214744 bytes)
18 .dynamic     000001a0
19 .data        00003558
20 .branch_lt   00000000
21 .opd         0002c778 (182136 bytes)
22 .got         0001cb50 (117584 bytes)
TOTAL:         514464 bytes
```

In other words, those three sections stand for the bulk of the extra space used by the 64-bit binary. And they are all parts of the ABI that's unique to the 64-bit version:

.opd contains the function descriptors .got is the global offset table .data.rel.ro is the data relocation information, and is moved there from the .rodata section used in 32-bit. This explains why the .rodata section is footnote-sizeer in the 64-bit binary.

3.2 x86

Looking at gcc 4.0.1's cc1 for i686 target:

64-bit:

```
text  data  bss  dec  hex
4889133 16520 562056 5467709 536e3d
```

32-bit:

```
text  data  bss  dec  hex
4330757 10112 507588 4848457 49fb49
```

Major differences are in data and bss:
260554 bytes (13%) more text
54468 bytes (11%) more bss

It's interesting to see that the amount of increase in bss is in the same ballpark as PPC (56692 bytes). It's likely that most of the increase is because of pointer sizes increasing, so going from 32 to 64 bits would give similar size changes on the two architectures.

Text size increase is also similar between the two, in both cases being a bit over 10%. Note however, that since x86 has a variable instruction length, a larger text segment does not necessarily mean more instructions in the code path. This can be verified by doing "objdump -d cc1 | wc -l":

32-bit: 1054860 lines

64-bit: 1006115 lines

Most of the decreased instruction counts are likely because of register argument passing, and the decreased overhead of that. If nops and data16 instructions are ignored, since in many cases they're never executed and just used for padding, the difference is even larger:

32-bit: 1033564 lines

64-bit: 894867 lines

4 Constants

4.1 PPC

The PPC instruction set only allows for loading of immediate values of 16 bits. On 32-bit values, it's normally done in one of two ways:

```
lis    rx,<upper half>
ori    rx,rx,<lower half>
```

or:

```
li     rx,<lower half>
oris   rx,rx,<upper half>
```

The difference is minimal. The 's' in the opcode indicates 'shifted'. In other words, "load immediate shifted" or "or immediate shifted", in either case the immediate value will be padded with 16 bits of 0 to the right.

If you remember from the look at global variables, the compiler can take a shortcut when loading an address to a variable that's being read or written, since the lower 16 bits can be used as an offset when loading the value, instead of initializing the register to the full 32-bit value and using offset 0.

For 64-bit constants, the same instructions must be used, there are no instructions to load into the top 32 bits in a register. So the code will be:

```
li     rx,<bits 16-31>
oris   rx,rx,<bits 0-15>
rldicr rx,rx,32,31
oris   rx,rx,<bits 32-47>
ori    rx,rx,<bits 48-63>
```

rldicr means "rotate left double word immediate clear

right”, and will rotate rx left by 32 bits and clear everything right of (but not including) bit 31.

This is 150% more code than on 32-bit, and all the operations are dependent, meaning that the result of the previous one must be available before next can be executed.

The positive side of this is that in normal code, the amount of 64-bit constants is normally quite low, and in cases where variables are initialized to them it’s done by putting the variable in the initialized data segment instead.

4.2 x86

Loading of constants on x86 is much less complicated than on PPC. The instructions are of variable length, so moving up to 64 bit values just means longer (new) instructions:

```
mov    $val,%target
```

In other words, 32- vs 64-bit x86 has no difference in number of instructions to load a constant of the native size.

5 Function call overheads

One area where both PPC and x86 has differences between the two ABIs is when it comes to function call conventions. On x86, the main difference is in the way arguments are passed, while on PPC, some of the differences are the introduction of function descriptors and TOC pointers.

Let’s look at a simple piece of C code. It’s probably not the most realistic testcase compared to what real software would do, but it shows the differences nicely.

```
int f(int (*fp)())
{
    return fp();
}
```

I.e. just a function that takes a function pointer and returns whatever the function itself returns.

The assembly for this is vastly different on 64-bit versus 32-bit, on both architectures.

5.1 PPC

64-bit:

```
.f:
    mflr 0          # save link reg in r0
    std  0,16(1)    # and save it away
    stdu 1,-112(1)  # buy a frame (112 bytes)
    std  2,40(1)    # store TOC ptr on stack
    ld   0,0(3)     # get entry point from
                    # from descriptor
                    # passed in
    ld   11,16(3)   # ???
    mtctr 0         # function addr in CTR
    ld   2,8(3)     # load new TOC
    bctrl                    # call function
    ld   2,40(1)    # restore TOC
    addi 1,1,112   # pop off stack frame
    ld   0,16(1)   # restore link register
    mtlr 0         # ..
    blr                    # and return
```

32-bit:

```
f:
    mflr 0          # Save link reg in r0
    stwu 1,-16(1)   # buy a frame (16 bytes)
    mtlr 3          # Move function pointer
                    # to link register
    stw  0,20(1)    # Store old LR on stack
    btrl                    # call function
    lwz  0,20(1)    # Load old link register
    addi 1,1,16     # pop off stack frame
    mtlr 0         # restore link register
    blr                    # and return
```

It can be seen that doing a simple function-pointer call on 64-bit is more complicated than on 32-bit, mostly because of the function descriptor reading and TOC management needed. It can also be seen that the footnote-sizeest stack frame is 112 bytes, even if only 8 of them were actually used (to save the previous TOC pointer).

So, what is the execution performance like for the two cases? Here we’re up for some surprises!

The testcase that was built did the above call one million times, timing the total time for all calls. The function pointer passed in was to a dummy function always returning 0.

Timing was made in timebase ticks, i.e. an external clock provided by the platform to the processor. On

the system in question, the frequency of the timebase is 33.333MHz.

```
64-bit: 351000 timebase ticks
32-bit: 755000 timebase ticks
```

It's more than half as slow on 32-bit, even though the function there is much simpler! What's going on??

In this case it comes down to the microarchitecture of the CPU. The PPC970 processor has a prediction cache for where a branch-to-link-register will take it, with the assumption that most branches to a link register is indeed a return to a previous function. In other words, it will push and pop values off of a stack of predicted values as "branch and link" and "branch to link register" instructions are executed. However, since the 'f' function in 32-bit will use the link register to do the call to the passed function pointer, the prediction stack ends up out of sync.

If the code is changed (by hand in the assembly) to instead use the CTR register, time used will be in the range of 333000 timebase ticks, i.e. slightly faster than the 64-bit binary, as could be expected given the slightly shorter code path. Also, keep in mind that since the same function is called over and over, the same function descriptor is referenced in the 64-bit case, resulting in always-cached contents. If the call would have been more random, and/or more spaced in time, the overhead of referencing the function descriptor would be a bigger factor.

During normal compiled code that doesn't use function descriptors, the behaviour would of course be different. For example, for calls to functions in the same module, the TOC normally does not need to be changed. Also, the loader would edit the branches to contain the direct address of the function, so there wouldn't be a need for run-time lookups of function pointers.

Since the TOC might need to be changed also for library calls, the compiler will always insert a NOP after an external call. This is so the loader can insert a function to restore the TOC value to point to the local one after the function return, since it's the responsibility of the caller to do so. Likewise, the called function will need to setup it's TOC pointer before using it to refer to other functions or global data.

Summary: With similar code being used, 64-bit code has about 6% overhead in function-pointer calls. For regular calls, the same overhead is in the range of 3%

5.2 x86

64-bit:

```
<f>:
xor    %eax,%eax # zero out eax
mov    %rdi,%r11 # function ptr to r11
jmpq   *%r11d    # jump to function
```

32-bit:

```
<f>:
push   %ebp      # Save ebp
mov    %esp,%ebp # Save old stack ptr
sub    $0x8,%esp # buy a stack frame
call   *0x8(%ebp) # call function
mov    %ebp,%esp # Restore esp
pop    %ebp      # and ebp
ret    # return
```

Big difference also on x86, the major one being the stack-passed parameters in the 32-bit ABI, making it necessary for even a footnotesize function to buy a stack frame. The 64-bit ABI allows for passing of parameters in registers, which makes it possible to just jump to the passed-in function pointer (and let that return directly to the original caller). Note however, that while the 32-bit ABI moves more data to and from the stack, the stack will very likely always be in L1 cache at the time the called function accesses it, making the real-life performance better than it could seem at first glance.

6 Benchmarks

To compare some of the theoretical differences between 32- and 64-bit binaries, a number of testsuites were used.

Hardware used was a Apple PowerMac G5, with 2 2.0GHz PPC970 and 1.5GB RAM and a whitebox Athlon X2-3800 system (nForce4 motherboard) with 1GB RAM. Both systems run with 64-bit kernels, only the userspace is different between the 32- and 64-bit environments.

While the results are presented here, a couple of observations were also made while preparing for the benchmarking.

One of the testcases (kernbench) needs a toolchain that will be the same for both environments. The easiest way

to achieve this is to use the crosstool[2] suite to build toolchains for the same targets.

To do this, all files were first downloaded and copied to both environments, to keep variations in network load and throughput from affecting the results. The native compilers in both environments were the same version (Gentoo GCC 3.4.4-r1) and the time it took to build the crosstool setups in the two environments was timed:

Building 64-bit crosstool on PPC64 for an i686 target took 51m 56s. Building the identical toolchain in the 32-bit environment took 46m 25s. In other words, the overhead of building in the 64-bit environment was 11%. On x86 the difference was less, 26m 5s in 64-bit and 26m 19s in 32-bit, a 7% difference with 64-bit being faster.

6.1 lmbench

lmbench[3] is a microbenchmark used mainly to measure the performance of the hardware and operating system it's being run on. It includes tests such as null syscalls, memory bandwidth and latency tests and basic integer and float operations. It also includes measurements of context switch times, fork times, file operations and network latencies and bandwidth.

Since the kernel and hardware was the same for 32- and 64-bit environments, quite a few of the tests showed little to no difference in performance between the two binaries. However, signal handler installation and invocation seemed to be about 10% slower on 32-bit, while some other cases were considerably faster in 32-bit. For example, exec was 42% slower on 64-bit, and executing a shell process was 36% slower.

Also context switches were faster on 32-bit, but variation of the timings made it hard to put a reliable metric on the results, ranging from almost no difference to 20% on the larger processes.

On File and VMM (normally mostly influenced by kernel-side performance), no significant differences between the two was observed, with one exception: Page fault handling was, for some reason, about 10% slower on 32-bit.

6.2 kernbench

Kernbench[4] is an ad-hoc benchmark of measuring the time it takes to build a specific version of the linux kernel sources with a specific configuration file and target architecture.

While it can be argued that compiling kernels isn't the most important usage for an operating system, it is still one of the more computation-heavy activities that many kernel developers use their systems for, the author of this text included.

Measuring the time it takes to build an i686 defconfig kernel with an x86 crosstool-build toolchain in both environments give the following results:

64-bit PPC:

```
Elapsed Time 253.47
User Time 452.784
System Time 49.934
Percent CPU 198
Context Switches 35381.8
Sleeps 28684.6
```

32-bit PPC:

```
Elapsed Time 208.69
User Time 378.95
System Time 34.812
Percent CPU 198
Context Switches 28363.2
Sleeps 24668.6
```

Difference in elapsed time and user time are about the same, 20% slower on 64-bit. However, the increased system time and context switch rate in the 64-bit environment is an interesting observation. The increased amount of context switches together with the increased amount of sleeps could indicate that more I/O is taking place, but in both cases all kernel sources should have been fully cached in memory. At this time, there's no better explanation available.

64-bit x86:

```
Elapsed Time 126.67
User Time 219.252
System Time 27.288
Percent CPU 194
Context Switches 16110.8
Sleeps 20814
```

32-bit x86:

Elapsed Time 162.612
 User Time 293.356
 System Time 29.108
 Percent CPU 198
 Context Switches 18534.4
 Sleeps 23238.4

rsa 2048 bits	37.8	1223.5	31.9	1102.7
rsa 4096 bits	5.5	339.4	4.8	319.5
dsa 512 bits	1442.2	1171.5	1237.2	1042.0
dsa 1024 bits	453.4	371.4	390.2	321.8
dsa 2048 bits	130.0	106.5	117.1	98.3

On x86, it's obvious that for compilation, going 64-bit gives a significant performance boost – 32-bit is 29% slower than the 64-bit compilation. What is interesting is to see that the difference in compilation time for kernbench is so significant, while the difference when building the toolchain was much less. The main difference between the two is that during toolchain build, output was a 64-bit binary, which might have made a difference.

6.3 OpenSSL speed test

The OpenSSL[5] package comes with a built-in speed test function that can be used to compare the performance of many commonly used hashes and signature algorithms.

	x86 64-bit	x86 32-bit
type	bytes/sec	bytes/sec
md2	4999.85k	6094.85k
mdc2	7211.69k	5057.19k
md4	424042.50k	362039.98k
md5	249194.65k	363050.33k
hmac(md5)	251434.33k	364980.91k
sha1	151977.98k	250208.26k
rmd160	108778.84k	130274.65k
rc4	159604.74k	188844.71k
des cbc	42027.69k	53612.69k
des ede3	16013.86k	18324.50k
idea cbc	42997.08k	34302.63k
rc2 cbc	22162.09k	21230.93k
rc5-32/12 cbc	107298.82k	176559.45k
blowfish cbc	76171.95k	88367.10k
cast cbc	57101.78k	38365.87k
aes-128 cbc	96561.83k	38858.93k
aes-192 cbc	85838.51k	33491.94k
aes-256 cbc	77236.91k	29439.49k

	PPC 64-bit	PPC 32-bit
type	bytes/sec	bytes/sec
md2	3189.71k	3077.46k
mdc2	5073.05k	5477.72k
md4	163916.20k	179702.44k
md5	107396.85k	114308.44k
hmac(md5)	108412.00k	115026.60k
sha1	106414.35k	137797.63k
rmd160	45249.23k	51041.62k
rc4	172576.32k	229673.64k
des cbc	25550.33k	27093.67k
des ede3	9729.70k	10302.81k
idea cbc	27308.48k	31569.24k
rc2 cbc	13090.87k	14985.90k
rc5-32/12 cbc	44674.98k	56188.93k
blowfish cbc	45722.79k	55812.10k
cast cbc	32830.60k	46260.22k
aes-128 cbc	66994.77k	70727.00k
aes-192 cbc	59045.00k	63034.71k
aes-256 cbc	52730.90k	56923.48k

	PPC 64-bit	PPC 32-bit
rsa 512 bits	1165.0	11982.6
rsa 1024 bits	241.0	4150.1

	x86 64-bit	x86 32-bit		
sign/s	verify/s	sign/s	verify/s	
rsa 512 bits	3992.2	48111.7	1734.0	20372.4
rsa 1024 bits	1276.3	20887.9	392.0	7814.4
rsa 2048 bits	239.6	7642.1	71.1	2493.0
rsa 4096 bits	38.7	2387.9	11.1	732.8
dsa 512 bits	6574.9	5646.4	2224.8	1910.0
dsa 1024 bits	2673.4	2233.0	841.7	711.6
dsa 2048 bits	898.3	726.9	268.1	218.2

Most of these functions are pure integer operations, some of such a nature that having 64-bit registers could benefit performance quite a bit if the functions can be performed on a bigger chunk of data per operation. In general, most hashing algorithms run about 5% faster in the 32-bit environment. When it comes to RSA and DSA signing, the relationship is the opposite: 64-bit performs 5-10% better than the 32-bit counterparts.

This is a benchmark where better performance would have been expected out of the 64-bit implementation. The fact that there are no hand-optimized assembly implementations for either environment could explain why there's not as much of a difference as expected. With more careful tuning, 64-bit should have an advantage.

6.4 AIM

AIM7/AIM9 and REAIM are a set of different benchmarks that essentially measure performance of running a multi-user environment on a system.

Using the OSDL version of the AIM7 benchmark[6], the “short” and “alltests” workfiles were executed.

```
PPC:                jobs/minute
alltests 64-bit:    3343
short 64-bit:       263631
alltests 32-bit:    3422    (2% better)
short 32-bit:       280500   (6% better)

x86:                jobs/minute
alltests 64-bit:    419444   (2.5% better)
short 64-bit:       6884    (2% better)
alltests 32-bit:    409146
short 32-bit:       6726
```

7 Conclusion

Given the observed performance during several workloads, it is obvious that in the case of PowerPC, generally 64-bit userspace environments have significant overheads compared to their 32-bit counterparts. For workloads where performance is important, and where it is not helped by having more than 4GB of memory addressable, staying with a 32-bit userspace environment is a wise choice. There are however still cases where the 64-bit environment brings benefits that makes it worth to still use, either in case of functionality and improved resource limitations, or in a few cases because of performance benefits of having 64-bit registers.

The case for AMD’s 64-bit architecture is the very opposite: Because of some of the other improvements that AMD did at the same time as the 64-bit expansions, 64-bit environments have in general considerable benefits over their 32-bit counterparts. In this case, migrating over to 64-bit brings both the benefit of larger address spaces where needed, as well as overall performance boosts. In the testing, there were very few cases where the 32-bit implementation had significantly better performance than the 64-bit one.

8 Disclaimers

As with all testing and measurements, there might have been errors introduced. The author has tried to keep as fair and accurate environment as possible, running both environments on the same hardware using the same kernel. On both architectures, this was achieved by installing the 64-bit version of Gentoo 2005-1 natively, and the 32-bit version in a chroot environment on the same disks. Systems had enough memory to keep free memory available during almost all of the testing, and care was taken to make sure that one of the environments didn’t get benefits of having data already cached compared to the other.

The GCC bug that uses the link register to do branches has been fixed since the benchmarking was done.

9 Trademarks

Linux is a trademark of Linus Torvalds.

Other company, product or service names may be trademarks or service marks of others.

This work represents the view of the author and does not necessarily represent the view his employer or any other entity.

References

- [1] Taylor, Ian Lance. 2004. “64-bit PowerPC ELF Application Binary Interface Supplement 1.7”, <http://www.linuxbase.org/spec/ELF/ppc64/PPC-elf64abi-1.7.txt>
- [2] Kegel, Dan. “Building and Testing gcc/glibc cross toolchains”, <http://kegel.com/crosstool/>
- [3] McVoy, Larry. “Lmbench - Tools for Performance Analysis”, <http://www.bitmover.com/lmbench/>
- [4] Kolivas, Con, et al. “Kernbench”, <http://ck.kolivas.org/kernbench/>
- [5] The OpenSSL Project. “OpenSSL: The Open Source toolkit for SSL/TLS”, <http://www.openssl.org/>

- [6] Taylor, Ian Lance. 2004. "Update and improve the existing Open Source AIM 7 benchmark", <http://sourceforge.net/projects/re-aim-7/>