# Tricks of a Spec master

Michael Meissner
*IBM*
`meissner@linux.vnet.ibm.com`

## Abstract

The 29 programs that make up the spec 2006 benchmark suite are often used to compare systems performance in the real world, but these benchmarks each have different characteristics. In this lightning round talk, I will cover at a high level the performance characteristics of these benchmarks in terms of optimizations that GCC does. For example, some benchmarks are classic floating point applications and benefit from SIMD (single instruction multiple data) instructions, while other benchmarks don't.

## 1 Spec benchmarks

The **SPEC** organization has strict rules on doing runs and reporting benchmarks, and companies often have whole performance groups dedicated to getting the best numbers as a means of reporting how a particular platform performs compared to other platforms. Most of the **SPEC** benchmarks are collections of freely available softare at a particular point in time with a fixed workload.

When I joined IBM two years ago after working at AMD, I made the observation that I was at a new company, with a different processor, but that I was running the same benchmarks, and many of the things I noticed at AMD were just as relevant at IBM.

This talk explores some of the characteristics that I and other coworkers have noticed in running spec benchmarks. I use the spec runs as a guide towards tuning the PowerPC GCC compiler. The percentages listed in this talk are gathered from various runs over the last year or two, and come from various compilers. I hope it is useful of things to think about when tuning GCC to run on various real world application.

This talk is not a representation of official numbers for IBM platforms, nor is it the offical views of the IBM corporation.

### 1.1 Spec 2006 INT benchmarks

The 12 Spec 2006 integer benchmarks are:

| **SpecINT benchmarks** | | |
|---|---|---|
| 400.perlbench | 401.bzip2 | 403.gcc |
| 429.mcf | 445.gobmk | 456.hmmer |
| 458.sjeng | 462.libquantum | 464.h264ref |
| 471.omnetpp | 473.astar | 483.xalancbmk |

### 1.2 Spec 2006 FP benchmarks

The 17 Spec 2006 floating point benchmarks are:

| **SpecFP benchmarks** | | |
|---|---|---|
| 410.bwaves | 416.gamess | 433.milc |
| 434.zeusmp | 435.gromacs | 436.cactusADM |
| 437.leslie3d | 444.namd | 447.dealII |
| 450.soplex | 453.povray | 454.calculix |
| 459.GemsFDTD | 465.tonto | 470.lbm |
| 481.wrf | 482.sphinx3 | |

### 1.3 Base and peak runs

The **SPEC** organization classifies runs as **base** and **peak**.

- **Base** runs are meant to run the entire benchmark suite with a common set of optimization options. In addition, there are rules that prevent profiling, and should reflect a high degree of portability, safety, and performance. The idea is that these would be options you would use to build a project, but without doing extensive tweaking of options, training runs, or other highly tuned options.
- **Peak** runs are no holds barred, optimize to the maximum even if it may violate language standards. You can profile a benchmark with training runs, specify different options for each particular benchmark, link in libraries on a per-application basis.
- In terms of tuning the compiler, I tend to take a middle ground, in that I generally try to use the same option on each build, but I will use `-ffast-math`, `-O3`, and some other options.

- I do break with the same option rule for the *400.perlbench* and *433.milc* benchmarks because I use the `-fno-strict-aliasing` option for those benchmarks alone. Those benchmarks do not run in some cases without `-fno-strict-aliasing`. The **SPEC** committee has ruled for base runs, `-fno-strict-aliasing` counts as an optimization option, and not a portability option. I don't notice the problem when doing *power7* runs, but I have noticed it in doing *power6* runs. For an official result with GCC, we would need to use the option, at least for *power6*. Using `-fno-strict-aliasing` does hurt performance in some benchmarks.
- When I was benchmarking GCC 4.3 based compilers, I also omitted the `-ftree-loop-linear` option when building *464.h264ref*, since the 4.3 compiler generated an internal error. More recent compilers have fixed this bug, so I now use the option on all builds.
- The spec numbers for each benchmark are ratios compared to a run of a particular machine in 2006. If you look at the numbers, you will see a wide swing of values, where some benchmarks have radically improved with newer machines, while other benchmarks have less improvement.
- **SpecINT** and **SpecFP** are each calculated as the geometric mean of the benchmarks. This has the effect that if you find a new optimization that helps a single benchmark, it brings up the combined value a little, but it can be frustrating when you get a 25% gain in a benchmark and the **SpecFP** number only goes up by a percent.

### 1.4 Current options used on power7

The options that I use change over time, but in the fall of 2010, the options I currently use for *power7* are:

- `-O3`
- `-fpeel-loops`
- `-funroll-loops`
- `-ftree-loop-linear`
- `-fvect-cost-model`
- `-ffast-math`
- `-falign-functions=16`
- `-falign-loops=32`
- `-mveclibabi=mass`
- `-mcpu=power7`
- `-mrecip=rsqrt`

## 2 Reciprocal estimate

The PowerPC architecture has several instructions that give an estimate which can then be refined with multiple Newton-Raphson steps to recover the accuracy. Newer generations of the PowerPC architecture (power6, power7) provide higher accuracy so that only 2 steps of Newton-Raphson fixup are needed.

- `FRE`, `FRES`: Estimate *1/x* for double and single precision.
- `FSQRTE`, `FSQRTES`: Estimate *1/sqrt(x)* for double and single precision.

The main function (*inl1130*) in the **435.gromacs** benchmark has nine occurances of the single precision *1/sqrt(x)* in the inner loop. By using the `-mrecip=rsqrt` option that was added in GCC 4.6, it sped up the **435.gromacs** benchmark by 25%.

The **437.leslie3d** benchmark speeds up by 3% when using the plain reciprocal estimate instructions, but **450.soplex** slows down, so at present, I recommend of optimizing just *1/sqrt(x)*.

The x86 computers have similar estimate instructions for single precision (not double precision), and the GCC compiler has optimized this for some time.

The function (*inl1130*) also is rather register intensive, and the PowerPC compiler has to spill intermediate values to the stack because the compiler ran out of floating point registers (32). One of the optimizations we are considering in the future is to spill single precision values to the upper 32 registers available under *VSX* rather than save/restore them to the stack.

## 3 Pow 0.75

I looked at the profile information for the **410.bwaves** benchmark, and discovered it was spending 50% of the time in the `pow` function. I looked at the benchmark and saw that the source only had two forms of `pow` useage:

- **a**2** which the compiler optimizes into a multiply;
- **a**0.75** which the compiler did not optimize.

I submited a patch that causes GCC to optimize `pow (x, 0.75)` into `sqrt(sqrt(x)) * sqrt(x)` when `-ffast-math` is used for all ports that provide a `sqrt` instruction.

| Benchmark | Percent |
|---|---|
| 410.bwaves | +48% |

## 4 Vectorized libraries

The x86 port has had the `-mveclibabi=acml` and `-mveclibabi=svml` options for some time to use the optimized *AMD* and `Intel` math libraries that vectorize various math functions under the `-ffast-math` option. In 2010, I added `-mveclibabi=mass` option to the PowerPC port to access the MASS optimized libraries. I measured the performance of just using the *MASS* library for its faster version of the math functions, and the effect of vectorizing the math functions under *Power7 VSX*.

Currently, the compiler only optimizes calls to the elemental vector elements, such as `V2DF` or `V4SF`, but the optimized math libraries also have functions that take two pointers and a length, and I had meant to add a pass to GCC that would convert code like:

```
for (i = 0; i < n; i++)
  a[i] = sin (b[i]);
```

into the appropriate vector calls. This should provide more benefit since the optimized math libraries can do the appropriate scheduling of the loads and stores.

| Benchmark | Library | Autovect |
|---|---|---|
| 459.GemsFDTD | +8% | +8% |
| 465.tonto | +73% | +75% |
| 481.wrf | +64% | +72% |

## 5 Floating point multiply and add instructions

The *454.calculix* benchmark benefits the most from having the compiler automatically optimize `(a * b) + c` into a fused multiply/add operations and `1.0 - (a * b)` into a fused negative multiply/subtract operation.

| Benchmark | Percent |
|---|---|
| 454.calculix | +16% |

Due to a bug that we haven't tracked down yet, we need to wrap the *atan2* library because something isn't dealing with -0.0. It occurs from the fortran line:

```
tt=datan2(dsqrt(1.d0-cn*cn),cn)/3.d0
```

I wrap the *atan2* call using the options `-Wl,-wrap,atan2` and linking in the following code with all programs:

```
static double zero = 0.0;

extern double __real_atan2 (double,
                            double);

double
__wrap_atan2 (double x,
              double y)
{
  return __real_atan2 (x + zero,
                       y + zero);
}
```

## 6 Benchmarks that are vectorized

I've looked at the code that is vectorized on the **SPEC** 2006 benchmark on the power7, which includes vector `char`, `short`, `int`, `float`, and `double` operations. I see that a lot of benchmarks generate vector code in various places, but when I looked closer, I didn't see that much vector code in the hot functions. One of the functions *410.bwaves* has vector code in the hotspot functions, but is slower than the scalar version. None of the benchmarks have vector integer or vector single precision operations generated in hot functions. It is interesting that the three benchmarks that can benefit from vectorized math libraries, don't seem to have other vectorizable code in the hot functions. It may be that so much time is spent doing the math functions, that it swamps the other calculations.

| Benchmark | Percent |
|---|---|
| 410.bwaves | -10% |
| 436.cactusADM | +60% |
| 437.leslied | +28% |

## 7 Code alignment issues

One of the very frustrating issues we've been dealing with when comparing two compilers, is that for a few benchmarks, depending on exactly where the hot loop is placed, we will see big swings in performance. Just bumping up function and loop alignments blindly can result in other slow downs.

| Benchmark | Difference |
|---|---|
| 410.bwaves | +/- 20% |
| 450.soplex | +/- 6% |
| 456.hmmer | +/- 12% |

## 8 Shrinkwrapping

Shrinkwrapping is a term we use to describe functions that have a simple test around the function, and if the test fails, the rest of the function is not executed. If the test involves an argument passed in a register and is often false, and the function will needlessly save and restore any saved registers that were used when the test is true. On some of the PowerPC machines the processor will stall while waiting until the store queue is emptied before it can do the loads in some cases. The place that we noticed it was in the *453.povray* benchmark in the `pov::Ray_In_Bound()` function.

| Benchmark | Profile | Elapse |
|---|---|---|
| 453.povray | +/- 3% | +/- 6% |

The savings in cpu time for using shrinkwap on the `pov::Ray_In_Bound()` function is 3% and the total elapsed time difference is 6%. This indicates that the cpu is spending a lot more time page faulting or idling while doing the useless stores.

## 9 Benchmarks with hotspot functions

Some benchmarks concentrate most of their execution in a few functions, while others distribute their time over more functions. It is a lot easier to look at the hot functions to optimize them rather than trying to come up with an optimization that speeds up programs in general. Here is a list of the benchmarks with hot functions:

| Benchmark | Percent | # functions |
|---|---|---|
| 470.lbm | 99% | 1 |
| 436.cactusADM | 99% | 1 |
| 456.hmmer | 96% | 1 |
| 462.libquantum | 94% | 3 |
| 437.leslie3d | 93% | 7 |
| 459.GemsFDTD | 90% | 5 |
| 410.bwaves | 89% | 2 |
| 401.bzip2 | 86% | 5 |
| 473.astar | 85% | 3 |
| 429.mcf | 85% | 3 |
| 444.namd | 73% | 7 |
| 435.gromacs | 71% | 2 |

## 10 32-bit vs. 64-bit

On the PowerPC architecture, I see roughly a 10% overall drop in performance when I run the **SpecINT** benchmark in 64-bit mode compared to 32-bit mode. The **SpecFP** benchmark is roughly 1% difference. However, it isn't consistant. There are benchmarks with big drops when running in 64-bit mode compared to 32-bit mode, but other benchmarks that are faster.

There are many different reasons for this performance difference:

- 32-bit programs have smaller pointers and generally have smaller stack frames and structure alignments. This means that in general the data cache will be more effective for 32-bit programs than for 64-bit programs.

- The ABI (application binary interface) is different for 32-bit and 64-bit programs. Depending on the system, this can have either a positive or negative effect. For instance, on the x86, in 64-bit mode, instructions often times have the **REX** prefix set to address the registers added in 64-bit. This makes instructions somewhat bigger, and makes the instruction cache less effective.

- On the other hand, 64-bit programs usually have a single instruction to do 64-bit integer arithmetic, while 32-bit programs have to issue multiple instructions to do 64-bit integer arithmetic. This affects programs that use *long long* types in C or *INTEGER* in Fortran defaulting to 64-bit.

- Many programs need 64-bit addressing these days, and would be severaly hampered when run on a 32-bit system. This is a case where the real world is not necessarily in tune with the benchmarking world.

### 10.1 Benchmarks that are faster in 32-bit mode

Some of the benchmarks that are slower in 64-bit compared to 32-bit are:

| Benchmark | Type | Percent | Cause |
|---|---|---:|---|
| 429.mcf | int | -31% | Pointer chasing |
| 471.omnetpp | int | -23% | Pointer and calls |
| 464.h264ref | int | -18% | Pointer chasing |
| 483.xalancbmk | int | -12% | Calls and malloc |
| 473.astar | int | -10% | Pointer chasing |
| 445.gobmk | int | -10% | Calls |
| 401.bzip2 | int | -9% | |
| 416.gamess | fp | -9% | |
| 450.soplex | fp | -9% | |
| 458.sjeng | int | -9% | |
| 482.sphinx3* | fp | -6% | |
| 444.namd | fp | -5% | |
| 447.dealII | fp | -5% | |
| 403.gcc | int | -5% | |
| 400.perlbench | int | -4% | |
| 465.tonto | fp | -4% | |

* The sphinx3 run was with a GCC 4.4 based compiler, as it did not work correctly in GCC 4.6 when I did the comparison.

## 10.2 Benchmarks that are faster in 64-bit mode

Some of the benchmarks that are faster in 64-bit compared to 32-bit are:

| Benchmark | Type | Percent | Cause |
|---|---|---:|---|
| 462.libquantum | int | +4% | long long |
| 410.bwaves | fp | +14% | fortran int |
| 481.wrf* | fp | +13% | fortran int |
| 436.cactusADM | fp | +9% | fortran int |
| 453.povray | fp | +4% | long long |

* The wrf run was with a GCC 4.4 based compiler, as it did not work correctly in GCC 4.6 when I did the comparison.