Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Improving Software Floating Point Support

Nathan Sidwell
CodeSourcery Inc
nathan@codesourcery.com

Joseph Myers
CodeSourcery Inc
joseph@codesourcery.com

Abstract

GCC's runtime library contains a set of software floating point routines, to be used when the required operation is not available in hardware. These routines have not been significantly optimized, and software floating point performs more poorly than it could. We discuss various pitfalls in their implementation. The GNU C library, `glibc`, also contains software floating point routines, and those have been optimized reasonably well. We show performance numbers obtained from portions of the EEMBC benchmark running on two PowerPC systems comparing the routines from the two libraries. We discuss the incorporation of the `glibc` routines into GCC's runtime library, and show how to convert other backends to use the new `glibc` routines.

1 Benchmarks

Our initial goal was to improve the performance of a subset of EEMBC[1] benchmarks running on PowerPC 405 and 440 hardware without using floating point instructions. The EEMBC benchmarks consist of a few sets of tests targeted at particular application areas—automotive, office, networking, consumer, etc. We used a subset of the automotive, networking and consumer sections. The automotive suite is particularly floating point intensive. After obtaining baseline benchmark numbers, we profiled the suite and examined each test's profile. Nearly all of the automotive tests spent some time in floating point routines. Five of the 16 automotive benchmarks spent significant time in a few floating point routines. Table 1 tabulates the time spent by those benchmarks in the floating point routines. It tabulates every routine accounting for more than 2% of the total processor usage. As can be seen, the benchmarks that use floating point spent considerable time in the floating point library. The final column is the geometric mean of the of the fraction of execution time for those benchmarks that showed usage. Mathematically, that is not a robust calculation because of the arbitrary 2% cutoff. However, it does give a guideline as to which routines are important.

2 Floating Point Libraries

GCC contains an implementation of software floating point in `fp-bit.c` and associated files. These implement the regular IEEE 754[2] operations of addition, subtraction, multiplication, division, comparison and conversions. Through the use of macros, `fp-bit.c` is used to generate `float`, `double`, and `long double` routines. In this paper, `fpbit` refers to the combined `float` and `double` routines

Testcase / Routine	basefp01	matrix01	a2time01	tblook01	iirflt01	Geometric Mean
__floatsidf			63.1%	18.1%	55.6%	39.9%
__muldf3	29.0%	32.42%	6.4%	13.3%		16.8%
__divdf3	12.7%					12.7%
__pack_d	16.3%	19.70%	7.4%	16.8%	4.7%	11.3%
__unpack_d	12.4%	15.74%	6.6%	9.6%	3.5%	8.5%
_fpadd_parts	22.1%	20.52%		3.9%		7.7%
__pack_f				5.8%		5.8%
__subdf3		4.45%				4.5%
__extendsfdf2				2.9%		2.9%
__adddf3	2.4%					2.4%
__divsf3				2.2%		2.2%
Total	94.9%	92.83%	83.5%	72.6%	63.8%	

Table 1: Benchmark Profiles

of these files. In addition, libgcc2.c contains some conversion routines which are used in certain circumstances.

Our initial plan involved optimizing fpbit itself. There are a number of improvements that can be made, and we estimated they would probably give a factor of 2 speedup on some of the routines. However, it came to our attention that an alternative library had already been proposed. Torbjorn Granlund submitted ieeelib[3] some time ago, but it had never been integrated. ieeelib implements many of the ideas we had for fpbit. We experimented by using it for the benchmarks and found that it gave a speedup of around 25% on EEMBC. Integrating ieeelib would be a better way forwards than improving fpbit itself.

Following this, we realised that glibc[4] also contained software floating point routines. Again we experimented with a version of GCC containing those routines and found it gave an improvement of around 20%. Clearly ieeelib and glibc were both candidates for integration. There were a number of advantages of each library:

- ieeelib has a smaller footprint than glibc.
- glibc contains support for different rounding modes, including runtime selection of the rounding mode. (The benchmarks we performed hardwired the rounding mode, so the comparison was comparing like for like features.)
- glibc has support for floating point exceptions, even integrating these into the hardware, when that is feasible. (Again, we made the above measurements with this disabled.)
- Using glibc routines would reduce the number of different software floating point implementations in GNU software.

This last point was very attractive. Reducing the number of implementations of software floating point would reduce maintenance. As we discovered, by uncovering some bugs both latent and otherwise, writing correct floating point code is tricky. Therefore, having a common implementation would improve software quality, because if a bug was found in either `glibc` or GCC, the patch could be applied to both.

The size difference between `ieeelib` and `glibc` seemed disadvantageous to `glibc`. Table 2 shows the sizes of `fpbit`, `ieeelib` and `glibc` routines. As can be seen, the first two are in the same ballpark, whereas the `glibc` routines are much larger.

Analysis showed there to be two causes of this. Firstly `glibc` has separate addition and subtraction routines, and secondly its multiplication and division routines are larger. Both of these turn out to have the same cause, namely correct support for NaNs, rounding modes and exceptions. Even though we had disabled as many additional features as possible, their effects were still present. We thought that it would be possible to improve the `glibc` code size somewhat, but were not sure how far the tendrils of the optional features could be removed. In the worst case, we felt that on a modern system an additional 6–7K bytes is not as significant as it used to be.

Additional advantages of `glibc` are its control of rounding mode and support for exceptions. Indeed, it could provide dynamic control of the rounding mode, which is desirable in some contexts. Although we were not immediately concerned with this, we were sure that others would be.

We decided that merging the `glibc` routines would be a technically better solution, and chose to pursue it.

However, there was a license issue; `glibc` is licensed under the LGPL[5], whereas the compiler's floating point emulation routines need to be licensed with runtime exception. That is, although the implementation of the routines can be licensed under the LGPL, merely linking them into a program as part of GCC's runtime support should not bring that program under the requirements of the LGPL (of course, this would not invalidate any other reason why the (L)GPL might apply). As the FSF[6] is the copyright holder of both `glibc` and GCC, only they could make the decision to allow the runtime exception for the `glibc` routines. We presented the technical arguments to the FSF, and persuaded Richard Stallman to allow a change of license. The FSF approved the use of LGPL plus runtime exception for the copies in *both* `glibc` and GCC. This means that the source files can be identical in both places, rather than having to add the runtime exception license wording to only the GCC copies.

3 Unpacking IEEE Numbers

The primary failing of `fpbit` is in its packing and unpacking of floating point numbers. Nearly all its deficiencies are artifacts of how this is done.

All `fpbit` routines commence by fully unpacking the floating point number's mantissa, exponent and sign into separate fields of a structure. In addition they determine the number's category as one of zero, denormal, signalling NaN, quiet NaN, infinity or regular number. The unpacked exponent is unbiased and the unpacked mantissa has the implicit 1 bit inserted. Denormals are scaled to be consistent with the regular representation of $-1^S \times M \times 2^E$. Apart from clearly taking time, this unpacking has an immediate deficiency. Firstly it means the floating point routines use structures, and pass them by address, thereby forcing these parameters to be passed in memory with all the slowdown that entails.[1] A second deficiency

[1]GCC's structure splitting optimization is inapplica-

Library	`fpbit`	`ieeelib`	`glibc`
Text size (bytes)	10688	9284	16940

Table 2: Library Sizes on PowerPC 440

is more subtle. The complete categorization makes the floating point routines begin with several separate checks for the rare categories. For instance the code of `_fpmul_parts` (the core of the multiplication routine) begins with:

```
if (isnan (a)) ...
if (isnan (b)) ...
if (isinf (a)) ...
if (isinf (b)) ...
if (iszero (a)) ...
if (iszero (b)) ...
```

This is actually more checks than is immediately apparent, because `isnan` checks for both quiet and signalling NaN categories. Naturally these checks have to be done, but the IEEE encoding uses only two special exponent values (zero and all ones) to encode all of the non-normal numbers. Thus it would be possible to have an early check for the two special encodings, and then determine which specific non-normal encoding has occurred out of the mainline of the routine. In fact, in the case of `_fpmul_parts`, this separation of all the distinct special cases is not necessary because *all* the separate `if` bodies are identical, or nearly so! This is a classic case of optimizing for the rare condition.[2] Before we started investigating `ieeelib` and `glibc` we made a 0.4% improvement by adding `__builtin_expect` calls to the various `isnan` and `isinf` macros so the compiler can optimize the expected code path.

ble here, as the packing and unpacking routines are not inlined.

[2]Implementers of networking stacks have discovered that early and complete unpacking of packet headers is a pessimization for the same reasons.

4 Improvements to `glibc`

We made a number of improvements to `glibc` in order to bring its performance, where it was deficient, up to that of `ieeelib`.

4.1 Unpacking

Some `glibc` routines do partial unpacking to obtain an exponent, sign and mantissa, thereby not having the pessimization described in Section 3. The exponent encodes the special values of interest. Most do further classification, but using macros and separate local variables instead of functions and structures, and using `switch` statements to reduce the number of checks. Depending on the operation being performed, `ieeelib` is even more specialized; it might do a minimal unpacking. It also never bothers adding in the implicit one bit in its expanded form. This turns out to be significant for float conversion operations and addition and subtraction where `ieeelib` was much faster than `glibc`. We patched `glibc` to perform minimal unpacking in these cases and improved its performance to be similar to that of `ieeelib`.

4.2 Addition and Subtraction

As mentioned, `glibc` has separate addition and subtraction routines. Naively it would seem that they could be trivially combined. Unfortunately this turns out to be difficult to achieve, because of the need to generate the correct NaN value in certain circumstances. A set of lower level macros, which provide target specific features, are used to construct glibc's routines. In the case of addition and subtraction, the macros of interest are _FP_ADD_INTERNAL and _FP_CHOOSENAN. The subtraction routine inverts the sign of the subtrahend *unless* it is a NaN. Then it calls _FP_ADD_INTERNAL. Addition simply invokes _FP_ADD_INTERNAL. This would suggest a simple merging scenario, but unfortunately:

- There is an excess of state to simply call an underlying routine efficiently.
- _FP_ADD_INTERNAL takes an operation parameter so that target specific code can return a different NaN for each operation, in the case of an exception.

Removing the excess state could be achieved by not bothering to detect a NaN subtrahend, and simply inverting its sign in all cases. This would change the behaviour of 'F − NaN'; rather than returning 'NaN', it would return '−NaN'. For GCC's purposes, the operation parameter is unimportant, but removing it would probably break our requirement that the GCC copy be readily updateable from the glibc sources. It also appears that glibc itself only uses this operation parameter for the x86 and x86_64 targets, where the software floating-point code is not actually used. However, the same glibc code is used in the Linux kernel math emulation, where consistency with hardware choice of NaN is required. It is unfortunate that this single target family causes such difficulty. It is possible that the distinction is unnecessary in even these two cases, in which case the operation parameter could be removed, and much simplification achieved. We decided to leave this as an open issue, keeping the addition and subtraction routines separate.

4.3 Bit Shifting

One significant change we made to fpbit before we started porting glibc was to use __builtin_clz to find the most significant set bit in integer to floating point conversion routines. This yielded a 7% performance improvement on EEMBC. We made the same changes to glibc where we replaced hand coded asm inserts with __builtin_clz, leaving it to the compiler to determine the most efficient code sequence.[3]

4.4 Other Patches

As part of preparing the glibc code for integration into GCC, support was added for the new floating point functions that had been inserted into fpbit since 1999. Functions were changed to use typedefs, such as SFtype instead of float. Additionally many changes were made to reduce the number of compiler warnings generated by the code. Some of these resulted from the heavy use of macros in the glibc code where unreachable code caused unwanted warnings.

4.5 Bug Fixing

In addition to improving glibc's performance, we uncovered a number of implementation bugs. This bolstered our thesis that software floating point can be tricky, and using the same implementation in both glibc and GCC

[3]In general glibc contains a large number of assembly inserts for 'optimized' code sequences. These might have produced better code than GCC in the past, but we now find the compiler producing better sequences. Worse, we have discovered that an assembly insert might be subtly wrong in that it does not describe the side effects or constraints correctly, leading to incorrect code generation with GCC's better optimizers.

would improve both. We found these bugs both through code inspection and the use of testsuites. Firstly, the GCC testsuite found some problems with the `glibc` routines. Secondly, we used the `ucbtest`[7] testsuite, which is designed for checking awkward IEEE cases. The bugs found and fixed in `glibc` include the following:

- Undefined behavior involving signed integer overflow.
- Undefined behavior involving shifting integers by the width of their type.
- Conversion of `float` to `long long` could left shift by a negative amount.
- Conversion of `long long` to `float` used a macro on `long long` values that only worked correctly on values of size `_FP_W_TYPE_SIZE` (typically `sizeof long`).
- An off-by-one-error in integer to floating point conversion when the integer value had exactly one more bit than the number of floating point mantissa and guard bits. For example, converting 3×2^{26} to `float` yielded 2^{28}.
- Incorrect exceptions were set in various cases.

We also found and fixed bugs outside of `glibc`:

- In EEMBC—reliance on undefined behavior of out-of-range floating point to unsigned integer conversions.
- In `fpbit`—a latent bug in a previously unused function causing incorrect rounding.
- In `libgcc2`—conversions of `TImode` (128-bit) integers to floating-point values had fundamental bugs.

5 Results

The EEMBC benchmarks report a number of values for each test. The value we used to measure improvement was the number of iterations per second. Because EEMBC reports iteration times as an integral number of microseconds, precision is lost with that more obvious measure of speed. As all the different tests have not been weighted against each other, we used the geometric mean in order to give each test equal weighting.[4] As stated earlier, we restricted our measurements to the automotive, networking and consumer subsections of the EEMBC suite. For the 405 benchmarks we used `-mcpu=405 -O2` and for the 440 benchmarks we used the `-mcpu=440 -O2` optimization flags. In addition to the floating point changes, we improved `strlen` and 16 bit multiplication by adding support for additional instructions. These particular benchmarks do not appear to make use of those features, and we believe the entire performance improvement shown here is due to the software float changes. Table 3 enumerates the before and after iteration counts and the speedup achieved. *Note, these are not official EEMBC benchmark results, and may be used as a speedup guide only.* As can be seen, the most improved test case's performance increased by nearly 360%.

6 Using the `glibc` Routines

We have imported the `glibc` routines into GCC. The primary source for these routines remains `glibc`, and any fixes to GCC's copy

[4]Using an arithmetic mean would unfairly bias the work to improving the speed of the longer benchmarks. There is no evidence that the longer iteration times are anything other than an artifact of the particular test being performed.

		405			440 softfp		
Benchmark	Tests	Before	After	Speedup	Before	After	Speedup
`basefp01`	1	3226.5	8762.2	2.72	13205.7	35550.5	2.69
`matrix01`	1	20.1	44.5	2.21	78.2	183.8	2.35
`a2time01`	1	26264.0	115293.7	4.39	97561.0	448129.1	4.59
`tblook01`	1	11009.2	23158.3	2.10	40566.3	97924.0	2.41
`iirflt01`	1	19656.4	60975.6	3.10	73432.2	234521.6	3.19
Automotive	16	13361.2	18445.2	1.38	52301.0	73497.2	1.41
Consumer	5	29.7	29.8	1.00	106.7	108.7	1.02
Network	6	802.9	806.3	1.00	2390.7	2386.9	1.00
Combined	27	2308.1	2797.2	1.21	8366.4	10266.2	1.23

Table 3: Benchmark Numbers

needs to be sent upstream to `glibc`. Fortunately the sources are identical in both GCC and `glibc`, because of the identical license change in both places.

The integration of the `glibc` routines into GCC has been designed to make it easy to start using these routines for new targets. Whereas `fpbit` uses special case code in `mklibgcc.in`, `glibc` uses the existing target makefile fragment mechanism. GNU Make features are used in `t-softfp` to select the functions required on a given target. In addition to defining the variables used by `t-softfp`, a file called `sfp-machine.h` must be provided for each target. Initial versions of this file for many targets are already located in the appropriate `sysdeps/ARCH/soft-fp` directory of `glibc`.

A target may specify the floating point and integer modes for which functions are to be compiled. The conversions between floating point modes to support may also be specified. This allows for targets with some hard-float and some soft-float modes. For those, `glibc` code will be used for conversions between the hard-float and soft-float modes. In such a case, the `sfp-machine.h` file may define how to raise exceptions and determine the rounding mode for the soft-float modes in a manner consistent with the exception flags and rounding modes provided by the hardware. A case where this might be useful in future is to support the optional `__float128` type in the x86_64 ABI.

7 Future Work

The `glibc` routines offer the possibility of further improvements. As has been already mentioned, the dynamic control of rounding mode is possible, along with integrating the exception mechanism with that provided by the system's `glibc` `fenv.h` interface. The routines could be extended to support the `float128` type present in the x86_64 ABI.

Further investigation of the issues involved in merging the addition and subtraction routines could be done, thereby reducing the code footprint.

There currently remains some overlap between the operations provided in the `glibc` routines and those provided by `libgcc2`. The `glibc` routines replace the `libgcc2` routines. For a pure soft-float target, this is exactly what is desired, but for a target with hardware floating point, but supporting a variant soft-float ABI, the `glibc` routines would be used in both sets of multilibs. The `libgcc2` routines will potentially handle rounding and exceptions consistent with the hardware floating point. This can be solved by implementing the above mentioned rounding and exception control to the `glibc` routines, and expunging the `libgcc2` routines from GCC. This would continue the reduction in the number of different floating point routines.

Further details of the `glibc` routines and suggested further work are available on the GCC Wiki at `http://gcc.gnu.org/wiki/Software%20floating%20point`.

8 Acknowledgements

This work was sponsored by the PowerPC Licensing Team at IBM. We are grateful for the opportunity afforded to speed up this part of GCC's support across all architectures.

References

[1] The Embedded Microprocessor Benchmark Consortium, `http://www.eembc.org`.

[2] Standard for Binary Floating Point Arithmetic, ANSI/IEEE Standard 754–1985.

[3] New IEEE P854 emulation library, Torbjorn Granlund (`tege@swox.com`), `http://gcc.gnu.org/ml/gcc/1999-07n/msg00553.html`

[4] The GNU C Library, `http://www.gnu.org/software/libc`

[5] GNU Lesser General Public License, `http://www.gnu.org/copyleft/lesser.html`

[6] The Free Software Foundation, `http:www.fsf.org`

[7] Testing difficult cases of IEEE 754 floating point arithmetic, David G. Hough (`dgh@validgh.com`) et al., `http://www.netlib.org/fp/ucbtest.tgz`