FLECKmarks
Measuring Floating Point Performance using a **F**ul**L** IEE**E** **C**ompliant Arithmetic Benchmar**K**

Joseph D. Darcy and David Gay
{darcy,dgay}@CS.Berkeley.EDU

1. Abstract

The floating point standard IEEE 754 is widely implemented, but many of its capabilities are not well supported by software or hardware. Programming languages provide no standard method to access IEEE 754 features, and the hardware performance is uneven, some operations being several orders of magnitude slower under certain conditions. We propose language extensions to span IEEE 754, measure the current level of hardware support, and propose "FLECK," a new suite of benchmarks using IEEE 754 features. The measurements are made on four recent architectures and compared to the SPEfp_base95 ratings of the same systems.

2. Introduction

Since the birth of high level programming languages with FORTRAN in 1950's, numerical computation on floating point numbers has been an important concern of computer users. Building on FORTRAN, later languages, such as ALGOL 60, provided more formal descriptions of syntax and semantics of valid programs. However, due to the variety of architectures of the time:

> No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis [12].

Therefore, the same source program compiled and run on different architectures produced different output due to varying range, precision, and other properties of a particular floating point format. In such a heterogeneous environment, a reasonable approach to provide cross-architecture portability is for a programming language to limit expressiveness to operations common to all (or most) contemporary architectures.

In order to eliminate the diversity of floating point formats at the time, the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985) was proposed and accepted. Among the standard's design goals were to

> Encourage experts to develop and distribute robust and efficient numerical programs that are portable, by way of minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity. When restricted to a declared subset of the standard, these programs should produce identical results on all conforming systems [2].

Since its introduction, IEEE 754 has become universally available on all significant microprocessors for PC's and workstations (Intel x86 line and clones, Motorola 68000, Power PC, HP PA RISC, Sun SPARC, SGI MIPS, and DEC Alpha, among others). IEEE 754 sports numerous features advantageous to the numerical analyst, if somewhat esoteric to the more casual programmer. Although the standard's features of directed rounding, subnormal numbers, arithmetic on infinities and NaNs (Not a Number), and floating point exception handling are all useful and are all implemented on conforming processors, these features are not supported in current programming languages. Programming language support for IEEE 754 was discussed before the standard was adopted [5]. Even if not employed directly, many users could benefit from more efficient libraries written using IEEE 754's advanced capabilities [3]. Due to the ubiquity of IEEE 754, programming languages should now support IEEE 754 specific features not available under other floating point standards. IEEE 754 was designed to allow sophisticated numerical algorithms, programming languages should not hinder that effort by lack of expressiveness.

Manipulating floating point values are fundamental operations in most languages; convenient to express and understood by the compiler. Therefore, while accessing IEEE 754 features via a library call interface is possible, full integration into a language allows for better ease of use and potentially faster execution times due to more precise program analysis and optimization. Due to the popularity and wide use of the C programming

language, the proposed language extensions discussed assume C as the language base (although the extensions are applicable to other languages as well).

While the processors mentioned above all support IEEE 754, the quality of that support varies considerably. The ease of access to the IEEE functionality is in practice a language and operating system issue, this aspect is covered by our proposed language extensions. The differences in speed can be compared with a standard benchmark. The floating point part of the SPEC95 CPU benchmark [15], derived from actual programs, provides a popular way of comparing the floating point speed of different processors. However, the codes used in SPEC95 only exercise "traditional" floating point features.

To get a better view of the relative performance of the floating point implementations available today, we have measured the speed of the basic operations specified in the IEEE standard and collected some realistic benchmark programs that exploit the special IEEE 754 features. We call this benchmark suite *FLECK*, for **F**ul**L** IEE**E** 754 **C**ompliant Arithmetic Benchmar**K**. From the results on these benchmarks, we obtain a 'FLECKmark', which we compare to the SPECfp95 results.

We have conducted these measurements on four modern processors: a 64 MHz HP Precision Architecture 7100LC, a 167 MHz UltraSPARC-1, a 250 MHz Alpha 21164 and a 200 MHz Pentium Pro. The Alpha is particularly interesting as it has two levels of IEEE support: fast but partial, and full but slow (with software support). Our measurements all require the latter mode, while DEC's official SPEC95 results use the former.

Section 3 contains an overview of the IEEE 754 standard and of our proposed language extensions. Section 4 presents our benchmarks and discusses their results. It ends with a comparison of the algorithms used in the FLECK suite with equivalent algorithms that do not use the IEEE 754 features. Section 5 discusses related work and presents our conclusions.

3. Programming Language Extensions for IEEE 754 Arithmetic

3.1 Brief Description of an IEEE 754 Machine

Before discussing the language extensions or benchmarks, the features of IEEE floating point need to be briefly summarized. The standard [2] discusses these features in much greater detail and should be referred to for more complete explanations.

The general structure of an IEEE 754 floating point number is

$$(-1)^s \cdot 2^E \cdot \left(b_0 . b_1 b_2 \ldots b_{p-1}\right)$$

where s is the sign bit (either 0 or 1), E is an exponent between E_{min} and E_{max}, and each b_i is either 0 or 1. Additional values include a positive and a negative infinity, and at least one NaN.[†] NaNs are used to represent invalid values, such as 0/0 or the square root of a negative number. The exact sizes and ranges of E and p (the number of bits in the significand) are given in the standard for two formats; single precision (32 bits total, p=24 bits, E=8 bits, E_{max} =127, E_{min} =-126) and double precision (64 bits total, p=53 bits, E=11 bits, E_{max} =1022, E_{min} =-1022). Constraints are also given for two other formats, single extended and double extended. Double extended (at least 79 bits total, $p \geq 64$ bits, $E \geq 15$ bits, $E_{max} \geq 16383$, $E_{min} \geq -16382$) has hardware support on the x86 and 68000 lines of processors. Numbers where b_0=0 are called subnormals (or denormals). Subnormals are encoded with a special exponent value not between E_{min} and E_{max}. Therefore, since all "normal" numbers have b_0=1, b_0 is not explicitly stored in the single or double format and is thus known as the implicit bit. In the single and double formats, a number has a unique encoding. Special exponent values outside of E_{min} and E_{max} are used to encode positive and negative infinity, positive and negative zero, and NaN values.

The standard defines addition, subtraction, multiplication, division, square root and comparison operations. Rules are given for propagating NaNs and arithmetic on infinities, see the standard for details. The value of an operation can be affected by the current rounding mode. By default, the IEEE floating point number closest to the infinitely precise real result is returned. Other rounding modes return values rounded toward 0, toward -∞, or toward +∞. Each operation can, as a side effect, set one of five sticky bits representing the conditions division by zero, overflow, underflow, inexact (rounding), and invalid (operation resulting in a new

[†] There are actually two classes of NaNs, signaling NaNs and quiet NaNs. Signaling NaNs are not widely used, are not used in the language extensions, and will not be further discussed in this document.

NaN such as 0/0, if a NaN is an input argument the invalid flag is not set by that operation). These bits are sticky in the sense that they remain set until cleared by the user, which implies the bits can be tested and manipulated by the user. Alternatively, instead of having the flags set, for each of the five conditions, the programmer can specify a trap hander. If this trapping mode is being used, the trap handler functions as a subroutine, returning a value for the operation which caused the exception. The trap handler must be able to determine a variety of information about the machine state when the exception occurred, such as what kind of operation was being performed.

3.2 Language Extensions

We propose four language extensions for IEEE arithmetic: handling of special numbers, rounding mode specification, new comparison operators and exception support.

3.2.1 Special Numbers

There are three new constants: `nan`, `infinity` and `-0` (which is slightly different from `0` in IEEE 754). There are a number of standard functions to classify values (`isnan()`, `finite()`, etc.) and break them into their components (`copysign()`, etc.).

As shown in section 4.1 the relative performance of processors on IEEE 754 special values ranges over several orders of magnitude. Subnormals generally have the greatest penalty for use, and could cause significant performance degradation on some processors. Processors with slow subnormals provide a faster mode of operation where small values are instead flushed to zero. The Alpha has the additional property that the mere possibility of special values necessitates running in a degraded mode.

Programmers need to be able to specify exactly what policies a program uses so that bit for bit repeatability is possible across processors. However, when performance concerns are paramount, a slightly different answer is acceptable if it comes more quickly. To allow both portability and performance, the programmer can declare a *refinement* of the types of floating point values which may occur. Essentially, the programmer specifies which special values must be handled correctly. A programmer specifies either to use subnormals, to use flush to zero, or to use whichever policy is faster on that architecture. A promise of only having traditional floating point values would allow the Alpha's fast execution mode to be used.

3.2.2 Rounding modes

The standard defines four rounding modes which affect the result of an arithmetic operation: to closest (the default), toward 0, toward +∞, and toward -∞.

In the IEEE model, the rounding mode can be changed dynamically at runtime. To provide structured access to this feature, a new enumerated type, `rounding_mode`, and a new declaration `round`, are introduced. The `round` declaration takes an expression of type `rounding_mode` and sets the current rounding mode accordingly, the rounding mode influences the results of subsequent arithmetic operations. The previous rounding mode is restored once the scope is exited. The default rounding mode is round to nearest.

Rounding modes are useful for implementing interval arithmetic and other types of error analysis. Since the rounding mode must be switched frequently during such computations, syntactic sugar in the form of new operators are provided to control the rounding mode at a very fine granularity. The rounding operators, such as `+^` to add and round toward +∞, specify a particular static rounding mode, which can lead to more efficient code generation on certain architectures, such as the Alpha. A static rounding mode operator is not affected by `round` declarations.

3.2.3 Comparison Operations

The various comparison operations (equal, not equal, less than, less than or equal, greater than, greater than or equal) are familiar and easily understood for traditionally available floating point values. However, NaN values that do not compare against other numbers; a NaN is neither greater than, less than, nor equal to any value, including itself. A NaN is *unordered* compared to other numbers. Existing comparisons against NaN (except for `==` and `!=`) set the invalid flag. Thus, the inclusion of NaN introduces subtleties into the comparison operators, $a < b \equiv \neg(a \geq b)$ does not hold under IEEE arithmetic. To provide comparison operators that are the complements of one another, a new set of comparison operators are needed, comparison operators that are true if

the unordered relation holds between the two arguments. These new operators, indicted by a "?" appended to the existing operator (e.g. >=?), also do not set the invalid flag when comparing against NaN. The unordered relation is checked with the "??" operator.

3.2.4 Floating Point Exceptions

The standard calls for user level access to the "sticky" flags that are set as a side effect of numerical operations. Although each flag can have a (user-specified) software trap enabled or disabled, we do not allow users to write their own trap handlers. Instead, traps are disabled and the standard's default behavior is used.

The value of the condition flags may be checked using the construct in Figure 1. After the computation completes the on structure checks the status flags set during the computation and runs the appropriate code. If more than one flag is set, the first matching clause listed is run. All the variable bindings in the computation block are available in the on clauses.

```
{
computation
}
on
{
overflow {...}
underflow, invalid {...}
div_zero, inexact {...}
}
```

Figure 1—Sample syntax for checking status flags

4. Comparing IEEE 754 Implementations

We first measure the speed of the basic IEEE operations to get an accurate view of the costs of the various features on each architecture. We then present our FLECK benchmark suite which uses these IEEE features. This suite is mostly based on existing programs, so is not written with the language extensions of Section 3.2. We then compare the performance of two of the FLECK benchmarks to equivalent algorithms that do not use the IEEE features.

4.1 Basic Arithmetic Operations

To compare the IEEE implementations of the four processor implementations, we measure the speed of the basic arithmetic operations +, *, /, and square root on all classes of IEEE numbers: normal numbers, infinities, NaNs, and subnormals. We do not measure subtraction or comparisons, as these should behave like addition. We also measure the cost of changing rounding modes.

We benchmark each operation in a pipelinable and non-pipelinable case[‡]: the pipelinable case repeats the same operation each time, while the non-pipelinable case makes each operation dependent on the previous one. We thus get a measurement of the throughput and latency of each of these operations.

The measurements are made by executing each operation 1 billion times, with the loop containing the instruction in hand-written assembly code. Each instruction is executed 10 times within each loop iteration, the times reported include the loop overhead. In practice, this loop overhead is hidden inside the latency of the floating point operations and therefore does not affect the timing results. When the operation is comparatively slow, only 10 million are executed. The reported times are the sum of "user" and "system" time, as returned by the operating system.

Table 1 through Table 4 report the results in nanoseconds and cycles per operation for each processor. Figure 2 through Figure 5 presents the time per operation graphically. The results for the Alpha 21164 in Table 4 includes times for a "fast" mode which acts only on normal numbers when full IEEE support is disabled.

[‡]Square root was not measured in a pipelined case since it's implementation generally not pipelined. Nor was square root measured for the Alpha's fast mode since it is implemented via a function call.

Table 1—Times for floating point operations on a 64 MHz HPPA 7100LC

	Normal		Infinity		NaN		Subnormal	
Operation	ns	cycles	ns	cycles	ns	cycles	ns	cycles
pipelined add	15.75	1.0	15.78	1.0	13950	893	14614	935
add	32.24	2.0	32.07	2.0	14222	910	15526	994
pipelined multiply	31.38	2.0	31.83	2.0	14503	928	22725	928
multiply	31.38	2.0	31.83	2.0	14503	928	22725	1454
pipelined divide	240.09	15.4	79.85	5.1	13438	860	31427	2011
divide	240.42	15.4	79.79	5.1	13481	863	31298	2003
square root	239.17	15.3	80.25	5.1	13512	865	31702	2029

Table 2—Times for floating point operations on a 200 MHz Pentium Pro

	Normal		Infinity		NaN		Subnormal	
Operation	ns	cycles	ns	cycles	ns	cycles	ns	cycles
pipelined add	7.04	1.4	576.3	115.3	636.6	127.3	7.02	1.4
add	15.05	3.0	576.3	115.3	575.7	115.1	15.06	3.0
pipelined multiply	10.04	2.0	565.7	113.1	590.9	118.2	10.03	2.0
multiply	10.23	2.0	506.5	101.3	531.5	106.3	10.25	2.0
pipelined divide	185.73	37.2	643.5	128.7	610.9	122.2	185.73	37.1
divide	185.95	37.2	556.7	111.3	551.7	110.3	185.96	37.1
square root	341.51	68.3	555.8	111.2	550.6	110.1	341.52	68.3

Table 3—Times for floating point operations on a 167 MHz UltraSPARC

	Normal		Infinity		NaN		Subnormal	
Operation	ns	cycles	ns	cycles	ns	cycles	ns	cycles
pipelined add	6.02	1.0	6.01	1.0	6.00	1.0	15773	2629
add	18.02	3.0	18.03	3.0	18.03	3.0	15923	2654
pipelined multiply	6.01	1.00	6.01	1.0	6.01	1.0	43034	7172
multiply	17.91	3.0	17.9	3.0	17.90	3.0	38162	6360
pipelined divide	132.17	22.0	48.06	8.0	48.05	8.0	72874	12145
divide	132.17	22.0	48.06	8.0	48.06	8.0	79179	19197
square root	132.14	8.0	48.06	8.0	48.05	8.0	94157	15693

Table 4—Times for floating point operations on a 250 MHz Alpha 21164

	Normal (fast)		Normal		Infinity		NaN		Subnormal	
Operation	ns	cycles	ns	cycles	ns	cycles	ns	cycles	ns	cycles
pipelined add	4.05	1.0	22.72	5.7	3250	880	3423	856	6483	1621
add	16.22	4.0	23.13	5.8	7000	1750	6858	1715	13345	3336
pipelined multiply	4.48	1.1	22.92	5.7	4425	1106	4342	1085	8872	2218
multiply	16.12	4.0	24.68	6.2	8882	2220	8615	2154	17384	4337
pipelined divide	130.12	32.5	144.55	36.1	9753	2438	9477	2369	31728	7932
divide	149.98	37.5	157.37	39.3	9840	2460	9588	2397	31825	7956
square root	—	—	358.57	89.3	308	77	290	73	731	183

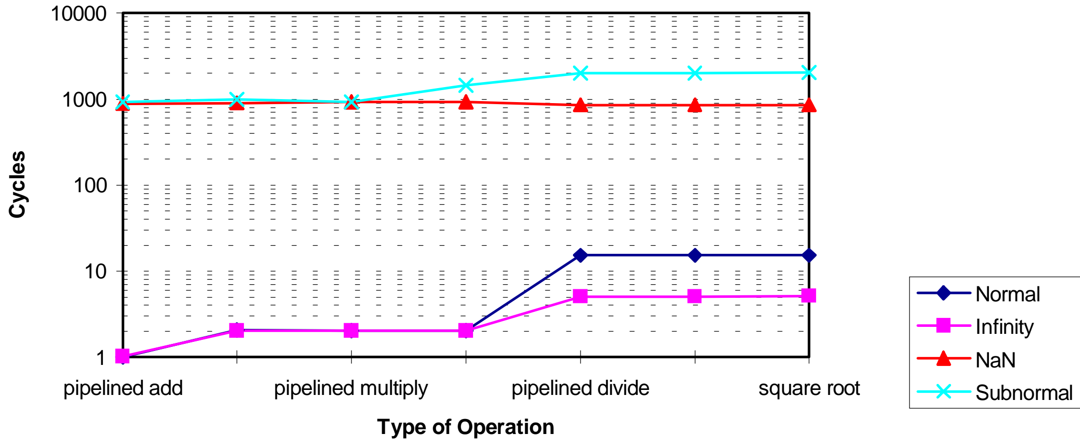

Figure 2—Information from Table 1 presented graphically

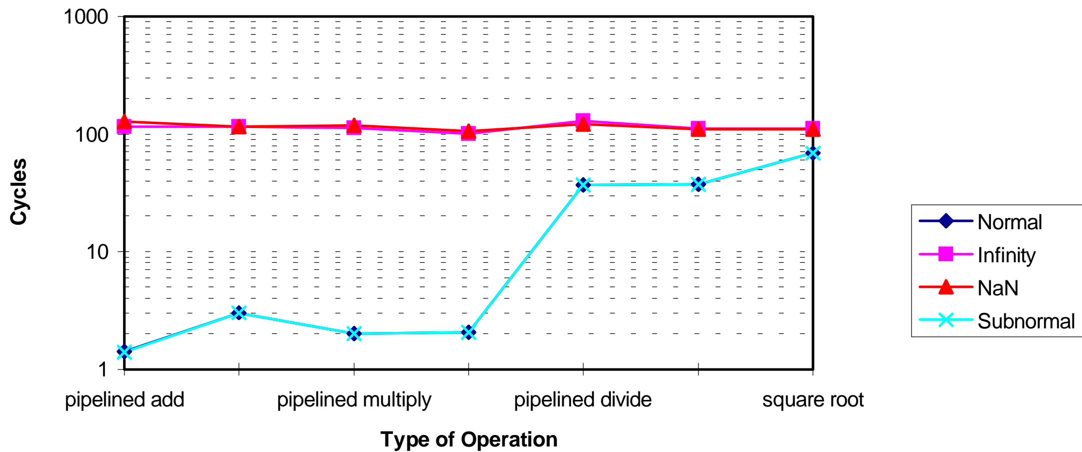

Figure 3—Information from Table 3 presented graphically

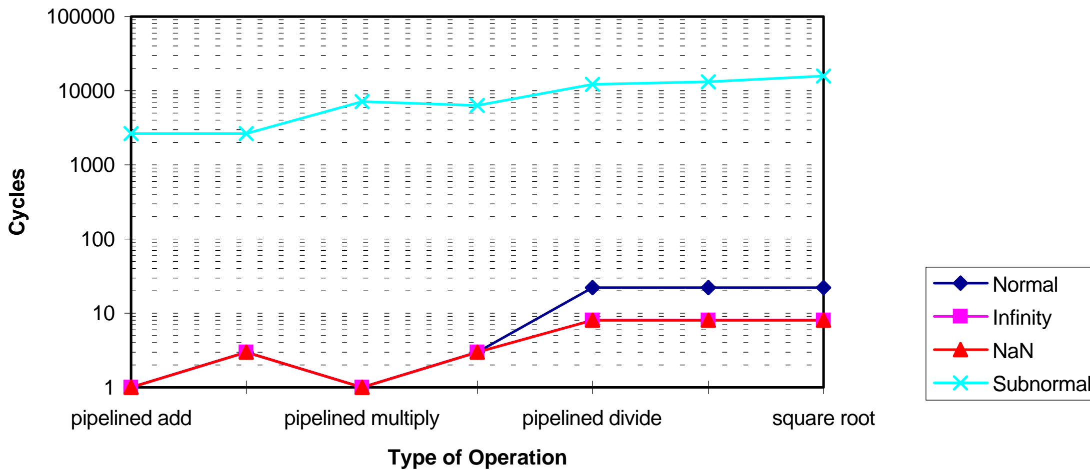

Figure 4—Information from Table 2 presented graphically

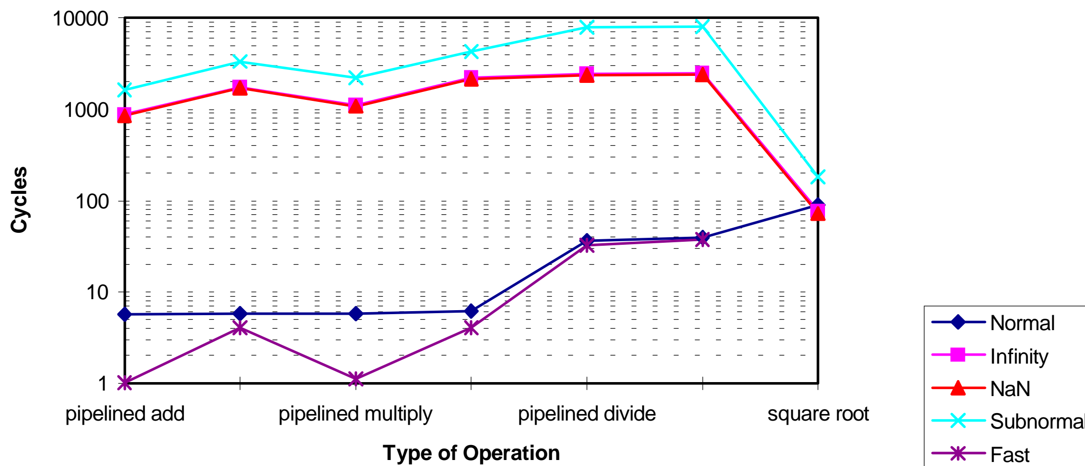

Figure 5—Information from Table 4 presented graphically

From these results, we can briefly summarize the weaknesses of each processor. The HPPA 7100LC has slow NaNs and subnormals, on the order of 1000 cycles per operation. The UltraSPARC runs all values except subnormals at full speed; unfortunately, when subnormals are used they are dramatically slower, with some operations taking over 10,000 cycles to complete. Finally, the Pentium Pro runs normals and subnormals equally well but suffers a penalty when operating on NaNs or infinities.

The stratification of performance on the Pentium Pro surprised us; we expected the Pentium Pro to have fast hardware support for the full IEEE standard. The slowdown is not as drastic as on the other processors, only one hundred cycles instead of thousands, but this can still impact performance, as the later benchmarks indicate. We surmise operations on NaNs and infinities are implemented in the Pentium Pro's microcode. To find out if the

slowdown of NaNs and infinities on the Pentium Pro is a new property of that member of the x86 lineage, we collected timings on a Pentium 166 and a 486 DX4 100. The Pentium has the same general performance profile as and Pentium Pro; NaNs and infinities were slow and subnormals were as fast as normal numbers. The Pentium Pro has somewhat better pipelining than the Pentium. The 486 does not pipeline floating point operations (an addition takes ten cycles), but multiplication and addition on NaNs and infinities are only about four times slower than for other values, taking about forty cycles total.

All the times on the Alpha, except for fast mode, are measured with a trap barrier between every floating point operation. While the architecture does not require quite that many barriers, this is the strategy used by DEC's compilers. This measurements reflect the performance that an application compiled with IEEE support enabled can expect. An attempt to reduce the number of trap barriers to one per basic block, which should be possible according to the Alpha architecture manual [13], produced speeds comparable to the "fast" results for normal numbers, but also gave incorrect results for the non-pipelined addition of subnormal numbers, which is rather unfortunate.

On machines implementing them in hardware, divides and square roots of infinities and NaNs are actually faster than normal values, presumably because the hardware aborts the operation. Similarly, the Alpha's software square root implementation allows it to have reasonably fast results for infinities, NaNs and subnormals, though it is substantially slower (both in time and cycles) than the HPPA 7100LC and UltraSPARC.

The cost of trapping to software to implement floating point operations varies significantly between the HPPA 7100LC, UltraSPARC and 21164 architectures and operating systems, with HP doing the best job, and Sun the worst. The Pentium's apparent trap to microcode is much faster in terms of cycles than any of the software traps by at least one order of magnitude.

To measure the cost of changing rounding modes, we cycle through each of the four rounding modes, doing one addition each time. We do 1 billion non-data-dependent additions. The results are reported in Table 5, in nanoseconds and cycles per change of rounding mode and addition.

Table 5—Times to change rounding modes and perform an addition on various processors

Processor	Time to Change Rounding Mode and Perform an Addition	
	ns	cycles
HPPA 7100LC	189.71	12.65
Pentium Pro	82.19	16.44
UltraSPARC	114.13	19.02
Alpha 21164 (allow dynamic)	52.70	13.18
Alpha 21164 (statically known)	23.32	5.83

The cycle-time results are all comparable, with the UltraSPARC being somewhat slower. The Alpha allows static specification of all rounding modes except one, which must be accessed via the dynamic rounding mode. In cases where static and dynamic rounding are mixed, the slower time is more realistic.

4.2 FLECK

This section presents the result of our program-level benchmarking. We first present the results of running the SPECfp95 benchmark suite and the Linpack [4] 1000x1000 benchmark to measure the performance of traditional floating point operations on our machine and compiler combinations. We then present our benchmarking suite, "FLECK," and report the results. We end with a comparison of all benchmark results.

We ran the SPECfp_base95 benchmark on each of our machine/compiler combinations so as to obtain SPEC numbers valid for our configurations.* All our benchmarks were run in multi-user UNIX configurations.

---

* On the Pentium Pro and Alpha, the benchmark 145.wave5 did not execute properly when compiled under the highest level of optimization. Therefore, it alone was compiled at the next lowest level of optimization. This procedure violates the rules for SPEC95 fp base which stipulate that all benchmarks must be compiled with the

On each architecture, all our benchmarks (except for the synthetic 'poly' benchmark) use the same compiler and compiler options. For the Alpha, we measure SPECfp_base95 with and without full IEEE support. The results are presented in Table 6. In each table we also include the official SPEC result for the machines closest to those we used. We normalize the SPEC numbers with respect to the HPPA 7100LC, our slowest machine.

Table 6—SPEC95 Float Base Information

Processor	HPPA 7100LC	Pentium Pro	UltraSPARC	Alpha 21164
Frequency	64 MHz	200 MHz	167 MHz	250 MHz
Compiler	HP f77	GNU g77	Sun f77	DEC f77
Compiler options	`+03` `+0nofastaccess`	`+03`	`-xO4` `-xdepend`	`-tune ev5` `-non_shared` `-om` `-O5` `-ieee_with_no_inexact` (not used for fast mode)
Operating System	HP-UX 9.07	Linux 2.0.0	Solaris 2.5.1	OSF1 V3.2
Measured SPEC95 FP Base	2.41	3.21	5.2	2.87 (fast) 7.05
Reported SPEC95 FP Base	2.66	5.99	8.45	8.39
Ratio of Measured to Reported	.91	.54	.61	.34 (fast) .84
Ratio of Measured to HPPA 71000LC	1.00	1.33	2.16	1.19 (fast) 2.93

Table 7 presents the MFlop rate on the Linpack 1000x1000 double precision benchmark, and the speedup relative to the HPPA 7100LC.

Table 7—Linpack double precision 1000x1000 performance

Processor	Linpack 1000 x 1000 Mflops	Speedup from HPPA 7100LC
HPPA 7100LC	9.344	1.00
Pentium Pro	13.09	1.40
UltraSPARC	20.37	2.18
Alpha 21164 fast	21.3	2.28
Alpha 21164	20.7	2.22

We wished to find benchmarks to measure the following aspects of the IEEE standard: special numbers (infinity, NaN, subnormals), exception handling, and rounding modes. Since we do not intend to measure the effects of the memory hierarchy or other non-processor system components, our benchmarks and datasets are fairly small. Our three benchmarks cover all these features except subnormals, we were unable to find an algorithm which depended on their presence for correct functioning (in fact, we found one, SDRWAVE [9], that depended on their absence). Two of our benchmarks are based on existing FORTRAN code, and one is written in C.

The two FORTRAN benchmarks are modified LAPACK [1] routines to get higher performance from exploiting IEEE features. They come from work of J. Demmel and X. Li [3].

---

same set of flags. However, to better compare to the Linpack and FLECK results, we did not degrade the optimization levels of the other nine benchmarks due to buggy optimizers on one program. Additionally, when compiling the Alpha in IEEE compliant mode, we used five flags, four for performance and one to specify IEEE compliance. Using more than four flags also violates SPEC95 base rules, but we wanted to get the highest base performance possible when running without IEEE support, so we used the four available flags and added the additional flag when using IEEE support.

The first benchmark, 'rcond', estimates the reciprocal of a matrix's condition number. Infinities and NaNs may arise during the computation, these are detected using the IEEE sticky exception flags. Rcond appears twice in the results, once with input that causes exceptions, once with input that does not.

'Eigen' is a FORTRAN program that computes the eigenvalues of a symmetric tridiagonal matrix. Infinities may arise during the computation, but these do not represent an "error" as they do in rcond. We also use two different inputs with eigen, one that causes infinities to appear and one that does not. The times reported for eigen and rcond are the time to execute the algorithm once, but the times were measured by executing them 100 or 1000 times to get more precise results.

The final benchmark, 'poly', computes an upper and lower bound for a polynomial at a given input by performing the same calculation under different rounding modes. This is a small synthetic benchmark. Both upper and lower bounds are calculated in each loop iteration, therefore, the rounding mode is changed in the inner loop. Another method to perform the calculation would be to run the code once to get the lower bound, change the rounding mode, and run again to get the upper bound. The changes to the rounding mode were done with inline assembly code to avoid the excessive cost of calling the vendor supplied functions to change the rounding mode. Table 8 shows the speedup achieved by using assembly code; these time are indicative of the times an IEEE aware compiler could generate. The Alpha assembly code was modified to use the explicit rounding modes allowed by that architecture (the static rounding mode row in Table 8). Since the static rounding mode is encoded as part of the instruction, the function call interface on the Alpha only implements fully dynamic rounding. Not surprisingly, if full dynamic rounding is used in the assembly code, the benefit of assembly code is lessened.

Table 8—Impact of using assembly code instead of function calls to manipulate rounding modes

Processor	Time using function call (sec)	Time using assembly (sec)	Speedup
HPPA 7100LC	223.22	29.28	7.62
Pentium Pro	22.84	11.35	2.01
UltraSPARC	28.87	14.09	2.04
Alpha 21164			
Static rounding	52.84	8.56	6.17
Fully dynamic rounding	—	20.58	2.56
Fast mode, static	—	1.93	27.38

Table 9 summarizes the execution times for these benchmarks. The speedups are given relative to the HPPA 7100 LC. The FLECKmark rating is the geometric mean of the speedups. Figure 6 presents these results in graphical form.

Table 9—Performance on FLECK

Benchmark	HPPA 7100LC		Pentium Pro		UltraSPARC		Alpha 21164	
	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup
rcond with exceptions	1834.1	1	157.13	11.67	14.16	129.56	1423.64	1.29
rcond, with no exceptions	88.40	1	40.04	2.21	40.12	2.20	30.72	2.88
eigen with infinities	180.86	1	167.54	1.08	116.50	1.55	131.16	1.38
eigen without infinities	181.45	1	130.09	1.39	116.98	1.55	129.65	1.40
poly	29280	1	11350	2.58	14090	2.08	8566	3.42
FLECKmarks		1		2.51		4.27		1.85

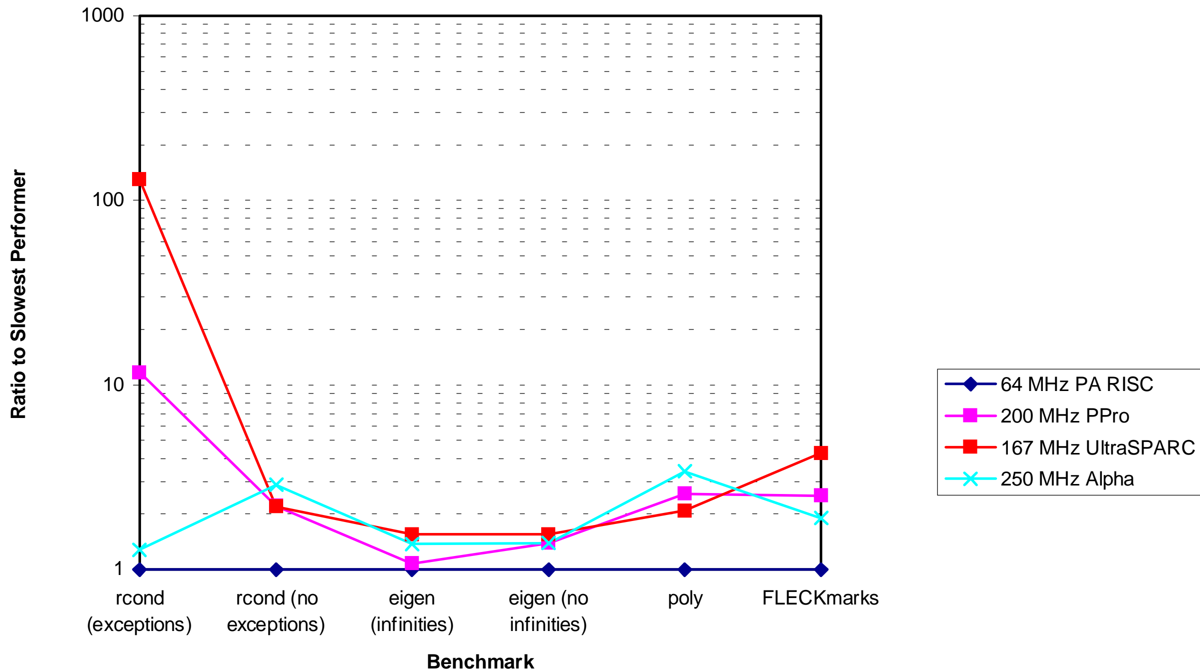

Figure 6—Performance on individual FLECK benchmarks and resulting FLECKmark rating

The performance differences on the input to rcond that has exceptions, and therefore infinities and NaNs, are very significant. This shows that the cost of not having hardware support for these numbers can be very high: the UltraSPARC is more than a hundred times faster than the HPPA 7100LC when exceptions occur, only two times faster when they do not. The Pentium Pro is an intermediate case here: as it's overhead for infinities and NaNs is much lower, it does a lot better than the 21164 or the HPPA 7100LC, but much worse than the UltraSPARC. When exceptional values do not occur, the 21164 is the fastest machine on this benchmark.

The eigen benchmark shows that some algorithms can rely on the support for IEEE features (infinities in this case) to get higher performance (see Section 4.3), without requiring that these features be implemented efficiently. Basically, eigen only generates a few infinities, so the performance of infinity arithmetic does not affect the result.

The results on the poly benchmark reflect the varying cost of changing the rounding mode, as shown in Table 5. Figure 7 compares the FLECKmarks to the SPECFP95 and Linpack results discussed above. It also shows the SPECFP95base results for machines similar to those we used in our tests.

The processors with the highest FLECKmark are the UltraSPARC and Pentium Pro, though this is mostly due to the exceptional input to the rcond benchmark. Except for this case, the 21164 does quite well on all benchmarks, even with full IEEE support enabled. All processors except the HP have a FLECKmark to normalized SPEC95 rating greater than one (see Figure 7), meaning that the HP performs worse on FLECK than its SPEC95 rating alone would lead one to believe.

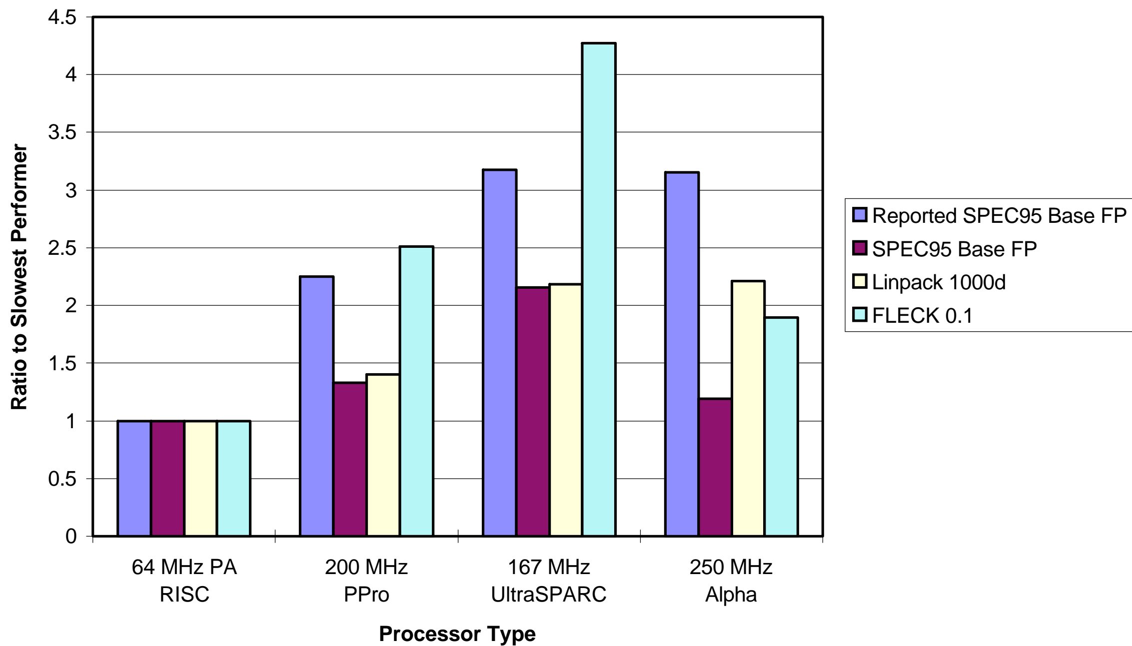

Figure 7—Relative processor performance on various benchmarks

4.3 Performance Benefits of IEEE

This section compares the performance of the algorithms used in the rcond and eigen benchmarks to the original algorithms from LAPACK used on the same inputs. It is thus a partial repetition of the results of [3] on more recent machines.

Table 10 lists the speedups for both benchmarks and both inputs when compared against the LAPACK version of the algorithm (which do not use any IEEE features). The IEEE features are accessed with function calls, so will be less efficient than in a language incorporating our proposed extensions.

Table 10—Ratio of execution time of algorithm not using IEEE to algorithm using IEEE

	rcond		eigen	
Processor	exceptions input	no exceptions input	infinities input	no infinities input
HPPA 7100LC	0.44	2.15	1.71	1.71
Pentium Pro	0.64	2.00	1.48	2.92
UltraSPARC	72.28	1..88	1.52	1.48
Alpha 21164	0.86	2.00	1.40	1.43

Even when many infinities and NaNs arise, the performance of the rcond IEEE algorithm is still competitive with the alternative LAPACK implementation (which has to do scaling to ensure the exceptions do not occur). When the hardware supports these features, the IEEE algorithms are always much faster than the originals.

5. Conclusions

5.1 Related Work

Other groups are also concerned with providing IEEE 754 floating point support in languages. The C9X group is working on incorporating floating point into the C standard [14]. Their proposal takes a less integrated approach than we recommend, relying on many macros and `#defines`. For example, instead of new constants, NaN and infinity values in C9X are returned by `NAN` and `INFINITY` macros. In C9X, the functionality of the new comparison operators discussed in Section 3.2.3 are implemented with function calls like `isgreaterequal()`. The working draft of the C++ standard [17] has facilities to query many properties of a floating point type relevant to IEEE floating point numbers. The proposed libraries for the functional language Haskell 1.3 include an optional library `LibIEEE_Float` that has rounding arithmetic operators and comparison operators similar to the structures in Sections 3.2.2 and 3.2.3.

In [8], W. Kahan discusses the features and implications of the IEEE 754 floating point standard as well as proposing language mechanisms to access those features. J. Hauser in [6] and [7] provides detailed examples justifying the usefulness of exception handling in floating point computations. Different language interfaces to exceptions are discussed in [7].

J. Demmel and X. Li measured the differing computation speeds on IEEE special values in [3].

5.2 Observations

Orders of magnitude difference in the speed of handling normal and special values exist on current microprocessors. The set of values computed fast is inconsistent across processors: the Pentium Pro does subnormals at full speed while all others take hundreds to thousands of cycles, the HPPA does infinity quickly but not NaNs while the UltraSPARC does both infinities and NaNs at full speed. These large differences are reflected in the FLECKmarks, which do not have the same ratios as the SPECfp_95 ratings. Providing hardware arithmetic on infinities and NaNs does not seem a priori hard, and can make a large performance difference in some algorithms (rcond). We therefore think that future processors should have this support.

Other algorithms are mostly insensitive to the performance of the basic operations (eigen), which shows that the IEEE features can always be useful. They should therefore have language support.

6. Acknowledgments

The authors would like to thank W. Kahan for his advice and discussion on matters related to this project, including recommending the poly benchmark. Alex Aiken provided feedback on the floating point language design issues. Xiaoye Li also deserves thanks for advice and assistance using the benchmarks from [3].

7. References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Gennbaum, S. Hammerling, A. McKenney, S. Ostrouchov, and D. Sorensen, LAPACK Users' Guide, Release 1.0 SIAM, Philadelphia, 1992.

[2] ANSI/IEEE, New York, IEEE Standard for Binary Floating Point Arithmetic, Std 754-1985 ed., 1985

[3] James W. Demmel and Xiaoye Li, "Faster Numerical Algorithms via Exception Handling", IEEE Transactions on Computers, Vol 43, NO 8, August 1994, pp. 983-992.

[4] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. LINPACK User's Guide. SIAM, Philadelphia PA, 1979.

[5] Richard J. Fateman, "High-Level Language Implications of the Proposed IEEE Floating-Point Standard," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 2, April 1982, pp. 239-257.

[6] John R. Hauser, Handling Floating-Point Exceptions, ACM Transactions on Programming Languages and Systems, Vol. 18, No. 2, March 1996, pp. 139-174.

[7] John R. Hauser, Programmed exception handling. M.S. Thesis, University of California, Berkeley, CA 1994.

[8] William Kahan, Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic, http://HTTP.CS.Berkeley.EDU/~wkahan/ieee754status/ieee754.ps

[9] David Kahaner, "Benchmarks for 'real' programs," SIAM News, November 1988.

[10] PA–RISC 1.1 Architecture and Instruction Set Reference Manual, Third Edition, Hewlett-Packard Company, 1996.

[11] Pentium Pro Family Developer's Manual, Intel Corporation, 1996.

[12] Naur, et al, "Report on the Algorithmic Language ALGOL 60

[13] Richard L. Sites, Richard T. Witek, Alpah AXP Architecture Reference Manual, Second Edition, Digital Press, 1995.

[14] Jim Thomas, C9X Floating Point, WG14/N595 X3J11/96-059 (Draft 9/12/96)

[15] Standard Performance Evaluation Corporation, http://www.specbench.org/

[16] David L. Weaver and Tom Germond, ed., The SPARC Architecture Manual, version 9, Prentice Hall, 1994.

[17] Working Paper for the Draft Proposed International Standard for Information Systems—Programming Language C++, Doc No: X3J16/95-0087