Cache-Optimised Methods for the Evaluation of Elementary Functions

David Defour

Laboratoire de l'Informatique du Parallélisme

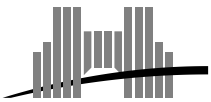
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

Cache-Optimised Methods for the Evaluation of Elementary Functions

David Defour

Octobre 2002

Research Report N° 2002-38



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

Cache-Optimised Methods for the Evaluation of Elementary Functions

David Defour

Octobre 2002

Abstract

The ratio between processor speed and memory speed frequently makes efficient use of cache memory a very important element in performance of user's application. This is the case for many elementary function algorithms. They commonly use tables, so that caches have a deep impact on speed. This paper discusses and quantifies the impact of cache usage over both major criteria in the evaluation of elementary functions: speed and accuracy.

Keywords: Elementary Functions, Cache memory, Microprocessors, Application Performance

Résumé

Cet article étudie l'influence de l'utilisation des mémoires caches sur les deux critères majeurs pour l'évaluation des fonctions élémentaires: vitesse et précision. En effet, nous observons que la différence croissante de vitesse entre processeurs et mémoires, rend l'utilisation efficace des mémoires caches importante pour obtenir des applications performantes. Les tests réalisés quantifient l'impact des caches de données sur la vitesse pour l'évaluation de fonctions élémentaires.

Mots-clés: Fonctions Élémentaires, Mémoire cache, Microprocesseurs, Performance des Applications

1 Introduction

The algorithms used for the evaluation of transcendental functions on modern processors follow the same rough scheme: reduction of the input argument / function approximation / reconstruction step. Such algorithms need to be portable, accurate and fast. The existence of a norm for floating point arithmetic is an invitation to develop an elementary function library geared toward portability. The precision we want to achieve is IEEE-754 double precision (including possible correct rounding). The speed of the implementation is commonly measured by the number of arithmetic operations that will be executed. However, most algorithms, [9], [10], [11], [12], use tabulated values. In this case, the speed estimation needs to include the access time to these tables, which cannot be neglected due to the increasing gap between memory and processor performance.

Hardware and software techniques help to reduce or hide the latency of main memory accesses. Software solutions, e.g. prefetch or alignment, can be used in some well defined cases to further improve cache performance. The classical hardware method of reducing the impact of memory latency is to use one or more levels of high-speed cache memory.

The introduction of caches inside the processor has a deep impact on speed of an algorithm. Although the number of operations performed in an algorithm can be determined easily, the cache influence over an algorithm is very difficult to measure. For this reason, the algorithms that we can find in the literature are not provided with their corresponding time estimations. For example we can find in [4] some number of tabulated values without any justification for this choice. On the other hand, when a time estimation is given, there is strong assumption about how the data are placed in the cache [5]. However all these assumptions do not provide an accurate measurement of the true impact of caches on function evaluations, and even less the applications using these functions.

This paper focuses on the speed of elementary function implementations. This will not only consider the time that takes the function routines to execute, but also the cost of a function call, and the influence of the cache misses that can be encountered. Therefore the estimation of speed that we give are closer to reality. We show that elementary function algorithms using less than 4 k.bytes will give the best performance on most architectures.

In Section 2 we describe how elementary functions are evaluated on modern processors. We then present in Section 3 modern processor unit's performances, whereas Section 4 discusses cache design and its influence on applications. In Section 5 we present the experiments done to test the influence of cache on speed and discuss the results. Finally Section 6 gives some concluding remarks, whereas the Annexe gives the details of the evaluation algorithms implemented for the tests.

2 Evaluation of elementary functions

We recall in this section the scheme that is most frequently used for evaluating an elementary function. More details can be found in Markstein [9] and Muller [10] books. We will give an example of evaluation with functions sine, cosine and logarithm in Appendix.

2.1 Target precision

The IEEE-754 standard requires that the results of additions, multiplications, divisions and square roots should be correctly rounded. But due to the difficulties of rounding elementary functions, there is no requirement concerning such functions. Elementary function implementations usually compute with more precision than the 53 bits of the double precision floating point numbers.

Therefore, when the evaluation is done with n extra bits, the probability to miss round to 53 bits is roughly proportional to $1/2^n$. To fix matters, we will aim in this paper at a precision of 70 bits of precision for the evaluation of elementary functions.

2.2 Evaluation scheme

There exist many ways to evaluate elementary functions. However in order to achieve 70 bits of accuracy, the most suitable method on modern architectures is composed of a range reduction, an approximation based on a polynomial evaluation, and a reconstruction.

Range reduction

For all elementary functions, the implementations have to provide a result for any input argument belonging to the set of all double precision numbers. It corresponds to numbers having an absolute value between 2^{-1074} and 2^{1024}. Unfortunately polynomial approximations to functions are accurate enough on a small interval only. The solution is to use some mathematical properties of the function being evaluated, to introduce a correspondence between the input argument and another number. This second number, called *reduced argument*, belongs to the interval where the polynomial evaluation is accurate. To achieve the precision of 70 bits for the result, some methods involve several range reduction steps making the range smaller and smaller. Beside reducing the polynomial degree, these several steps also reduce the use of time expensive multiple precision operators, in the subsequent polynomial evaluation scheme.

If the first range reduction step is done using an algebraic relation, the following ones are usually done by tabulating some values of the function. Tables are addressed by the first leading bits of the input argument.

Evaluation

To reach around 70 bits of accuracy, the evaluation methods are based on polynomial evaluations over a small interval. The coefficients of these polynomial approximations are usually determined by a Chebyshev approximation or with a Remes algorithm.

There exist several solutions to evaluate such polynomials [8]. The simplest is to use Horner's scheme. But in that scheme, each instruction depends on the previous one. So it leads to many stall inside the long pipeline of modern architectures. This poor performance of Horner's evaluation scheme can be improved by dividing the initial polynomial into two or more independent polynomials.

Reconstruction

The reconstruction step consists in collecting all the partial results that were obtained during range reduction and polynomial evaluation and put them together. These results are merged carefully to preserve the accuracy and keep a small dependency chain.

2.3 Parameters of this scheme

The previous description shows that many parameters need to be taken into account: the first parameter is the required accuracy for intermediate computations; afterward, the range reduction scheme including the tabular range reduction; and the polynomial approximation are adjusted to fit the precision and speed goal. In Section 4.3 we will discuss how these last two parameters can be tuned considering cache memory.

3 Arithmetic operations in current processors

The characteristics of the arithmetic operators of recent processors that are most useful for this study are given in Table 1.

Note that the division operator is used in some range reduction schemes [11], but due to its high latency, such algorithms are not very common. Low latency and throughput of integer units are used for comparisons to reduce stalling inside the pipeline due to branch or jump. On the other side, when we look for accuracy we use the high precision floating point units.

Design	Cycle (ns)	Integer			Floating-point			
		Nb. Units	$a \pm b$ L/T	$a \times b$ L/T	Nb. Units	$a \pm b$ L/T	$a \times b$ L/T	$a \div b$ L/T
Alpha 21264	1.6	4	1/1	7/1	2	4/1	4/1	15/12
Athlon K6-III	0.71	6	1/1	1/1	3	3/1	3/2	20/17
Pentium III	1.0	3	1/1	4/1	1	3/1	5/2	32/32
Pentium IV	0.4	3	.5/.5	14/3	2	5/1	7/2	38/38
Itanium IA-64	1.25	4	1/1	18/1	4	5/1	5/1	$>$36/$>$26
PowerPC 750	2.5	2	1/1	2-5/1	1	3/1	4/2	31/31
Sun UltraSparc IIi	2.27	2	1/1	4/1	2	3/1	3/1	22/22
Sun UltraSparc III	0.95	4	3/1	4/1	2	4/1	4/1	24/17

Table 1: Latencies (L) and throughput (T) in clock cycles of ALU and FPU of modern processors.

4 Cache memory in modern processors

Cache performances are crucial for many applications including elementary function libraries. Therefore we give in this section the background material about cache memory that is necessary for Section 5.

4.1 Memory hierarchy

The classical method for reducing the impact of memory latency is to use one or more levels of high-speed cache memory. Caches work by exploiting locality of reference, meaning that if some data is referenced once, then that item, or one near it in the address space, is likely to be referenced in the near future. Locality of reference is exploited by holding referenced data items in small, fast memory located close to the processor.

These caches can be *unified*, and they will be able to store instructions and data without any distinction. Otherwise there might be *separated*, and there will be split between data and instructions. In a typical code, instructions may refer to everything that is known at compilation time (as for example all the constants), whereas a randomly accessed table is considered as data.

The current trend is to have separated caches of level 1 between instruction and data. It is the case for all processors listed in Table 2.

4.2 Cache organization

Caches are divided into *blocks*, each block being composed of a group of consecutive instructions or data in the address space. There are several techniques to arrange cache for block placement

Design	Cycle (ns)	Level 1 Data Cache			Level 2 Data Cache		
		Size (ko)	Lat-ency	Nb Ways	Size (ko)	Lat-ency	Nb Ways
Alpha 21264	1.6	64	3	2	<16MB	12	
Athlon K6-III	0.71	32	2	2	256		
Athlon XP	0.55	64	2	2	256		
Pentium III	1.0	16	3	4	512	10	
Pentium IV	0.4	8	2	4	512	7	
Itanium IA-64	1.25	16	3	4	96	7	6
Itanium II	1.0	16	2	4	256	6	
PowerPC 750 (G3)	2.5	32	2	8	<1MB	13-20	
PowerPC 7455 (G4)	1.0	32	2	8	256	9	
Sun UltraSparc IIi	2.27	16	2	1	<2MB	6	
Sun UltraSparc III	0.95	64	2	4	<8MB	15	

Table 2: Main cache properties of modern processors. This table describes the size of the data cache, an integer load latency and the number of ways.

depending on where a block can be placed in the lower level. A cache is *direct-mapped* if each block has a specific place in the cache, where the place is usually computed by a simple modulo the number of blocks. Opposed to that, a cache is *fully-associative* if a block can go anywhere in the cache. And a trade-off between these two methods is a *set-associative* cache where a block can go anywhere in a subset of blocks in the cache. In that case the placement function is usually a simple modulo the number of sets in the cache. The number n of blocks in a set determines the number of ways. These caches are also called n-way set-associative caches. They are the most used in current processors for level 1 cache, as we can notice in Table 2. Therefore, in the rest of this paper we focus on exploiting properties of these caches.

In a n-way set-associative cache, when a cache miss is encountered, the data needs to be loaded from the upper level and then placed in one of n different places in the lower level cache. So new data will take the place of an existing one. Hence a replacement technique needs to be used. These techniques are LRU (least recently used), FIFO (First in first out), or random replacement.

4.3 Cache and applications

In a set-associative cache it may be difficult to predict, while programming, if declared data will go in the same subset of block in the cache. This problem is due to the techniques used by the memory management unit that uses mechanisms such as virtual address or page segmentation well detailed in [6] and [7]. But if we consider a table, due to the locality principle, two consecutive table data blocks will not go into the same subset. As a consequence, if the table size is less than a way size, this table will be stored in one way without any data block stored into the same subset. In such a scheme one way is used to store the table and the other ways are kept empty for other usages.

As we saw in Section 2 we need to perform intermediate computation with a precision higher than the one available in the hardware. In most cases, a tabular range reduction is needed. This range reduction, based on the use of a table, will lead to different performance in terms of time depending on how often the mother application calls the considered function.

Occasional calls to a function

When occasional calls to a function are done only, we can suppose nothing is in the cache. So the instructions and data used for the evaluation of elementary functions will need to be brought into cache level 1. In this case, the polynomial evaluation cost and the range reduction access cost will be unsignificant compared to all the cache miss costs. We give in Section 5 the results of experiments done to measure the influence of the table size used for range reduction versus the polynomial length over the speed for one function call.

Frequent calls to a function

When many calls to the same function are done, we can suppose that the first calls will bring and keep instructions and data in cache. If data and instructions are small enough to stay in the cache between two calls, we can expect good performance in subsequent calls. But since elementary functions are used in an application, to get good performances, there must also remain enough space in the lowest level of the cache for the instructions and data of the mother application.

As we saw in Part 4.2, current processors have data and instructions cache separated from cache level 1. Moreover they often are set-associative with an average size of 4 to 16 k.bytes per way. Based on that remark, and on the fact that data in a table have consecutive addresses, if the considered table exceeds a way size, we may encounter some replacements in some subsets of blocks between pieces of data of the same table. Hence when several calls to the considered function are done, some cache misses will occur. This will slow down the application. On the other hand, if the table is small enough to stay in a cache way, this problem will not appear. The following section will quantify this impact.

5 Experiments

We have made some experiments to check the impact of cache usage for the evaluation of elementary functions.

The goal is to provide the fastest implementation of elementary functions for a given accuracy. Moreover performance must be achieved for applications making occasional or frequent calls which should not slow them down.

As noticed in Section 2.1, the evaluation methods must provide 70 bits of accuracy. In addition, by choosing the method, we will exclusively measure the relative impact on speed of the variation of polynomial degree and the table size used for the range reduction.

5.1 Experiment description

To test the impact of table and polynomial size over speed and accuracy we have chosen three typical functions: the logarithm function because this function needs a high degree polynomial to be well approximated; and the sine and cosine functions because they are known to be well approximated by a low degree polynomial.

The trade-off between the cost of the tabular range reduction and the polynomial degree is given in Table 3 for the logarithm and in Table 4 for trigonometric functions. However these tables do not include the constants used inside the program nor the polynomial coefficients. Both may be stored in the program, and this uses instruction cache and no data cache. The polynomial degrees were chosen to reach at least 65 bits of precision for the logarithm and 70 for trigonometric functions. The coefficients of the approximations were obtained using the *numapprox* package of Maple [1]. The detailed evaluation scheme chosen is given in appendix A.

Programs were written in C and compiled with *gcc-3.0*. The execution times were collected by processor specific assembly instructions that read the tick counter. So all the times are given either in clock cycles (this is the case for the Pentium III and the Itanium), or in a unit that is a small multiple of clock cycle, approximately 3 to 10 times (this is the case for the UltraSparc IIi).

We have tested these functions on a Pentium III workstation based on a Celeron processor, on an HP workstation i2000 based on an Itanium processor and on a Sun Ultra 5 workstation based on an UltraSparc IIi processor. All the tests have been done on randomly generated values.

Polynomial degree	Polynomial accuracy	Table size in Bytes	k
20	65	48	2
15	66	96	3
12	67	176	4
10	68	368	5
8	65	720	6
7	66	1456	7
6	65	2896	8
6	71	5792	9
5	67	11584	10
5	73	23168	11
4	66	46336	12
4	71	92688	13

Table 3: Trade-off between the tabular range reduction and the polynomial degree required to reach 65 bits of precision over $[-2^{-k}, 2^{-k}]$ for the function $\log(1+x)$.

Sine poly. deg.	Sine poly. accuracy	Cosine poly. deg.	Cosine poly. accuracy	Table size in Bytes	k
7	82	7	78	512	4
6	78	6	74	1024	5
5	72	5	69	2048	6
5	81	5	77	4096	7
4	71	4	67	8192	8
4	76	4	73	16384	9
4	82	4	79	32768	10
3	70	3	68	65536	11
3	75	3	72	131072	12
3	80	3	77	262144	13

Table 4: Trade-off between the tabular range reduction and the polynomial degree required to reach around 70 bits of precision over $[-2^{-k}, 2^{-k}]$ for $\cos$ and $\sin$.

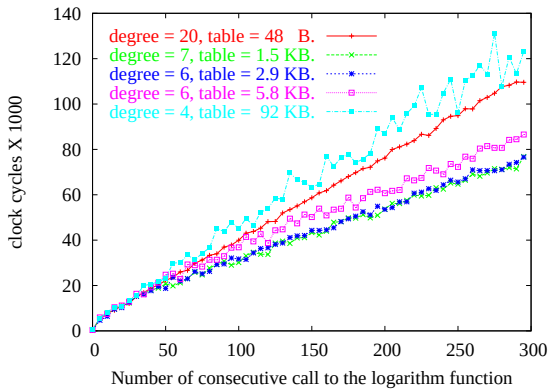


Figure 1: Time taken by the tested logarithm function on a Celeron processor.

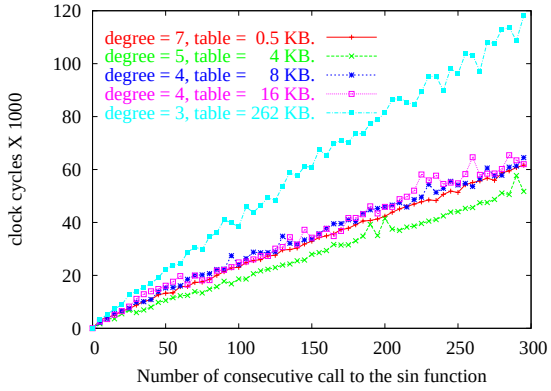


Figure 2: Time taken by the tested sine or cosine functions on a Celeron processor.

5.2 Experiment results

Results for occasional call to a function

Table 5 describes the number of clock cycles for one call to an elementary function depending on the architecture.

We remark that the obtained results are very irregular and not correlated to the table size nor to the polynomial degree of the evaluation scheme. This is because the time taken by cache misses dominates all the other execution times. This characteristic can be observed for all tested functions and on each processor. Therefore, for occasional calls to a function the degree of the polynomial approximation or the size of the table do not have a measurable impact on execution speed.

Results for frequent call to a function

The time taken to execute up to 300 consecutive calls to the logarithm and the sine or cosine functions are respectively recorded in Figures 1 and 2. Both graphs describe the execution time

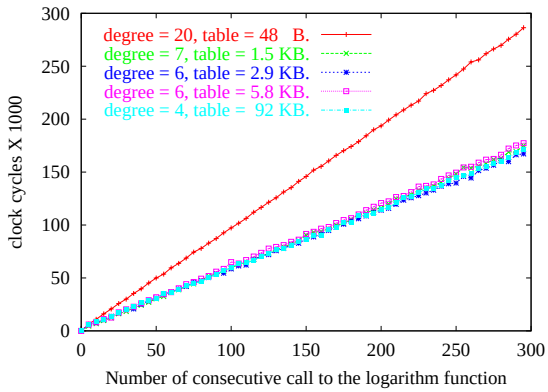


Figure 3: Study of time taken by the tested logarithm function on a SunUltra IIi processor.

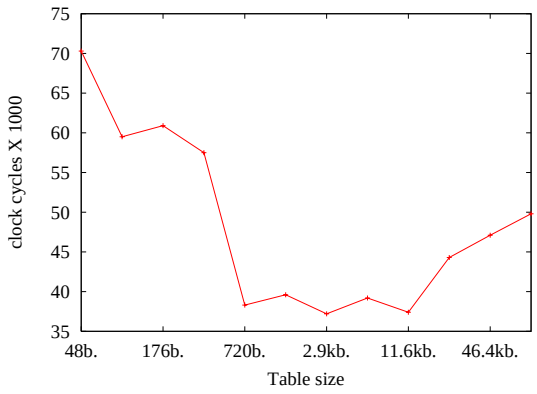


Figure 4: Impact on speed of the table size on an Itanium processor for the logarithm function.

on the Celeron processor, built with 4 k.bytes size per way in the data cache. We observe on these graphs that the execution time is roughly linear with the number of consecutive calls to an elementary function.

Differences between architectures

Graphs for the Itanium processor are very similar to the results obtained with the Celeron. The difference between the Itanium and Celeron results lies in the amount of difference between the best graph and the others. The higher number of way of cache, and its larger size for the Itanium processor make this difference less important. It must be noticed that floating point load/store are performed on this processor from cache of level 2 where the size of one way of data cache is 16 k.byte.

The Sun-Ultra IIi has a level 1 data cache direct-mapped of 16 k.bytes and a 2 MB level 2 cache. Results for trigonometric function are very similar to the logarithm results given in Figure 3. This graph shows that on this architecture with direct mapped data cache, the polynomial evaluation cost seems more important than the table size used in the range reduction. The small difference

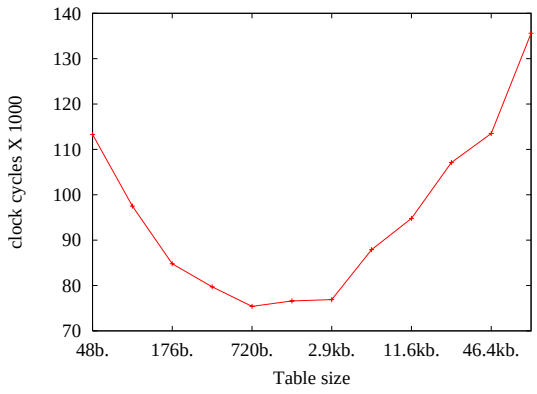


Figure 5: Impact on speed of the table size on a Celeron processor for the logarithm function.

	k	4	5	6	7	8	9	10	11	12	13
Pentium III	log	3610	3500	3222	3635	3953	3455	3404	3140	4526	3727
	trigo.	1307	1232	1688	2114	1386	1967	1412	2971	1127	1216
Itanium	log	1330	1308	1428	1321	1559	1362	1314	1374	1332	1338
	trigo.	1387	1151	1380	1120	1077	1030	1139	1050	1157	1005
SunUntra IIi	log	1961	2152	3321	1131	2848	3237	2477	2842	2711	2327
	trigo.	3200	3401	2617	3509	2655	2629	3559	3417	2565	1959

Table 5: Number of clock cycles for one call to an elementary function depending on parameter k (k is defined in Tables 3 and 4).

on this Figure, between tested methods is also due to the large cache of level 2 with small access time.

Differences between functions

On each tested type of architecture, we observe that the difference between methods is more important for the logarithm function than for the trigonometric functions. This is correlated to the larger difference in the polynomial degree for the logarithm than for the trigonometric functions. It means that for heavy use of a function, the number of floating point operations in the polynomial evaluation has an impact over speed. For example let us consider Figure 1, that describes the difference between methods for the evaluation of the logarithm. In this graph, there is a 32 percent time difference between the method using a degree 20 polynomial approximation and the one using a degree 7 polynomial approximation.

Cache misses impact

On each figure we observe that the graphs corresponding to small table size are more regular than the graphs representing methods with large table size. This phenomenon is due to the largest number of cache miss that might be generated for large table size. Therefore for a large numbers of calls to a function, methods with small table size are preferable because there will present more regular results.

Size way impact

Except for the SunUltra IIi processor which is direct mapped, we notice on both tested functions that best performance is achieved for methods using a table size slightly less than a way of cache.

This remark also holds in Figure 4, that compares the time taken to evaluate 300 consecutive logarithms on an Itanium architecture. This graph shows that best performance is achieved for methods using table size less than a way size (methods with 11.6 k.bytes table size). In addition we observe on the left side of the graph the influence over the number of clock cycles of the polynomial evaluation and the related number of floating point operations. Contrary to the right side of the graph where the execution time is dominated by the number of cache misses. All this can also be observe on in Figure 5.

6 Conclusion

We have shown that best performance is achieved if the table size does not need more than a way size cache where floating point load / store are made from. Furthermore, we saw that contrary to the heavy use, for occasional use of a function, no measurable difference exists between each implemented version.

Mathematical libraries are bound to work on a large variety of processors where the average size of a data cache way is 4 to 16 k.bytes (Table 2) and this is true even within a family of processors (e.g. x86). Then, the efficiency of mathematical libraries can be improved, if the developed functions that will often be called together (such as sine and cosine) use less than 4 k.bytes.

Future work will include real-world tests, and especially the interaction between tables used for each transcendental function.

References

[1] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Library Reference Manual*. Springer Verlag, Berlin, Germany, 1991.

[2] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[3] W. J. Cody. MACHAR: A subroutine to dynamically determine machine parameters. *ACM Transactions on Mathematical Software*, 14(4):301–311, December 1988.

[4] S. Gal and B. Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, March 1991.

[5] J. Harrison, T. Kubaska, S. Story, and P.T.P. Tang. The computation of transcendental functions on the IA-64 architecture. Technical Report RR2001-26, Intel technology Journal Q4, 1999.

[6] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, pages 60–75, July 1998.

[7] B. Jacob and T. Mudge. Virtual memory: Issues of implementation. *Computer*, 31(6):33–43, June 1998.

[8] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

[9] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[10] J.M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[11] P. T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378–400, December 1990.

[12] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.

A Description of evaluation scheme

A.1 Logarithm

The logarithm implementation follows the evaluation scheme described in Section 2. A special case detection, to detect overflows, underflows, NANs and all other special value is the first operation performed. Then a range reduction is executed to get a reduced argument belonging to the range $[-\sqrt{2}/2, +\sqrt{2}/2]$. Let $x = 2^E(1+f)$ with $0 \leq f < 1$ be the input argument, then this first range reduction is based on the following mathematical formula

$$\log(x) = (E+1)\log(2) + \log(\frac{1+f}{2})$$

In order to provide an efficient evaluation scheme, this first range reduction step needs to be completed with another range reduction step based on tabulation of some values. Let $w_i = 1 + i2^{-k}$ be the closest number to $1+f$, then this second range reduction is based on the formula

$$\log(1+f) = \log(w_i) + \log(1 + \frac{1}{w_i}(1+f-w_i))$$

with

$$|\frac{1}{w_i}(1+f-w_i)| < 2^{-k}$$

This requires tabulation of values $\log(w_i)$ and $\frac{1}{w_i}$ since the division is an expensive operation (Table 1). The number of tabulated values is then equal to $2(1-\lfloor(\sqrt{(}2)/2)2^k\rfloor 2^{-k})$ double precision numbers.

The trade-off between the tabular range reduction and polynomial degree for 65 bits of precision is summarized in Table 3.

A.2 Trigonometric functions

Unlike the logarithm function, trigonometric functions have the property of being well approximated by a low degree polynomial. As for the logarithm all the implementations were based on the same scheme, only the polynomial evaluation differs. The first step is a special case detection followed by a range reduction based on Cody and Waite's method [2][3]. For an input argument x in double precision, the selected method computes the integer $N = x\frac{2^k}{\pi}$. Then the reduced argument r is defined as

$$r = (x - NC_1) - NC_2$$

where C_1 is an approximation of $\frac{\pi}{2^k}$ in floating point double precision and C_2 is and approximation of $\frac{\pi}{2^k} - C_1$. Since $r \in [-\pi/2^k, +\pi/2^k]$ we have

$$\sin(x) = \sin(r)\cos(N\pi/2^k) + \cos(r)\sin(N\pi/2^k)$$

where $\cos(N\pi/2^k)$ and $\sin(N\pi/2^k)$ are 2^{k+1} tabulated values and $\sin(r)$ and $\cos(r)$ are computed by polynomial approximation. The link between the parameter k used for range reduction and the degree used for polynomial approximation is summarized in Table 4.

Based on the identity $\cos(x) = \sin(x + \pi/2)$, the same function is used for the cosine where the difference is in the addition of 1 to the parameter N. This evaluation scheme is also the one implemented in the IA-64 mathematical library from Intel [5].