The Computation of Transcendental Functions on the IA-64 Architecture

John Harrison, Microprocessor Software Labs, Intel Corporation
Ted Kubaska, Microprocessor Software Labs, Intel Corporation
Shane Story, Microprocessor Software Labs, Intel Corporation
Ping Tak Peter Tang, Microprocessor Software Labs, Intel Corporation

Index words: floating point, mathematical software, transcendental functions

ABSTRACT

The fast and accurate evaluation of transcendental functions (e.g. exp, log, sin, and atan) is vitally important in many fields of scientific computing. Intel provides a software library of these functions that can be called from both the C* and FORTRAN* programming languages. By exploiting some of the key features of the IA-64 floating-point architecture, we have been able to provide double-precision transcendental functions that are highly accurate yet can typically be evaluated in between 50 and 70 clock cycles. In this paper, we discuss some of the design principles and implementation details of these functions.

INTRODUCTION

Transcendental functions can be computed in software by a variety of algorithms. The algorithms that are most suitable for implementation on modern computer architectures usually comprise three steps: reduction, approximation, and reconstruction.

These steps are best illustrated by an example. Consider the calculation of the exponential function exp(x). One may first attempt an evaluation using the familiar Maclaurin series expansion:

$$\exp(x) = 1 + x + x^2/2! + x^3/3! + \ldots + x^k/k! + \ldots .$$

When x is small, computing a few terms of this series gives a reasonably good approximation to exp(x) up to, for example, IEEE double precision (which is approximately 17 significant decimal digits). However, when x is large, many more terms of the series are needed to satisfy the same accuracy requirement. Increasing the number of terms not only lengthens the calculation, but it also introduces more accumulated rounding errors that may degrade the accuracy of the answer.

To solve this problem, we express x as

$$x = N \ln(2) / 2^K + r$$

for some integer K chosen beforehand (more about how to choose later). If $N \ln(2)/2^K$ is made as close to x as possible, then $|r|$ never exceeds $\ln(2)/2^{K+1}$. The mathematical identity

$$\exp(x) = \exp(N \ln(2) / 2^K + r) = 2^{N/2^K} \exp(r)$$

shows that the problem is transformed to that of calculating the exp function at an argument whose magnitude is confined. The transformation to r from x is called the reduction step; the calculation of exp(r), usually performed by computing an approximating polynomial, is called the approximation step; and the composition of the final result based on exp(r) and the constant related to N and K is called the reconstruction step.

In a more traditional approach [1], K is chosen to be 1 and thus the approximation step requires a polynomial with accuracy good to IEEE double precision for the range of $|r| \le \ln(2)/2$. This choice of K leads to reconstruction via multiplication of exp(r) by 2^N, which is easily implementable, for example, by scaling the exponent field of a floating-point number. One drawback of this approach is that when $|r|$ is near $\ln(2)/2$, a large number of terms of the Maclaurin series expansion is still needed.

More recently, a framework known as table-driven algorithms [8] suggested the use of $K > 1$. When for example $K=5$, the argument r after the reduction step would satisfy $|r| \le \log(2)/64$. As a result, a much shorter polynomial can satisfy the same accuracy requirement. The tradeoff is a more complex reconstruction step, requiring the multiplication with a constant of the form

$$2^{N/32} = 2^M 2^{d/32}, \quad d = 0, 1, \ldots, 31,$$

*All other brands and names are the property of their respective owners.

where $N = 32M + d$. This constant can be obtained rather easily, provided all the *32* possible values of the second factor are computed beforehand and stored in a table (hence the name table-driven). This framework works well for modern machines not only because tables (even large ones) can be accommodated, but also because parallelism, such as the presence of pipelined arithmetic units, allow most of the extra work in the reconstruction step to be carried out while the approximating polynomial is being evaluated. This extra work includes, for example, the calculation of *d*, the indexing into and the fetching from the table of constants, and the multiplication to form $2^{N/32}$. Consequently, the performance gain due to a shorter polynomial is fully realized.

In practice, however, we do not use the Maclaurin series. Rather, we use the lowest-degree polynomial *p(r)* whose worst deviation $|p(r) - \exp(r)|$ within the reduced range in question is minimized and stays below an acceptable threshold. This polynomial is called the *minimax* polynomial. Its coefficients can be determined numerically, most commonly by the Remez algorithm [5].

Another fine point is that we may also want to retain some convenient properties of the Maclaurin series, such as the leading coefficient being exactly 1. It is possible to find minimax polynomials even subject to such constraints; some examples using the commercial computer algebra system Maple are given in reference [3].

DESIGN PRINCIPLES ON THE IA-64 ARCHITECTURE

There are tradeoffs to designing an algorithm following the table-driven approach:

- Different argument reduction methods lead to tradeoffs between the complexity of the reduction and the reconstruction computation.
- Different table sizes lead to tradeoffs between memory requirements (size and latency characteristics) and the complexity of polynomial computation.

Several key architectural features of IA-64 have a bearing on which choices are made:

- Short floating-point latency: On IA-64, the generic floating-point operation is a "fused multiply add" that calculates $A \acute{} B + C$ per instruction. Not only is the latency of this floating-point operation much shorter than memory references, but this floating-point operation consists of two basic arithmetic operations.
- Extended precision: Because our target is IEEE double-precision with 53 significant bits, the native 64-bit precision on IA-64 delivers 11 extra bits of accuracy on basic arithmetic operations.
- Parallelism: Each operation is fully pipelined, and multiple floating-point units are present.

As stated, these architectural features affect our choices of design tradeoffs. We enumerate several key points:

- Argument reduction usually involves a number of serial computation steps that cannot take advantage of parallelism. In contrast, the approximation and reconstruction steps can naturally exploit parallelism. Consequently, the reduction step is often a bottleneck. We should, therefore, favor a simple reduction method even at the price of a more complex reconstruction step.
- Argument reduction usually requires the use of some constants. The short floating-point latency can make the memory latency incurred in loading such constants a significant portion of the total latency. Consequently, any novel reduction techniques that do away with memory latency are welcome.
- Long memory latency has two implications for table size. First, large tables that exceed even the lowest-level cache size should be avoided. Second, even if a table fits in cache, it still takes a number of repeated calls to a transcendental function at different arguments to bring the whole table into cache. Thus, small tables are favored.
- Extended precision and parallelism together have an important implication for the approximation step. Traditionally, the polynomial terms used in core approximations are evaluated in some well specified order so as to minimize the undesirable effect of rounding error accumulation. The availability of extended precision implies that the order of evaluation of a polynomial becomes unimportant. When a polynomial can be evaluated in an arbitrary order, parallelism can be fully utilized. The consequence is that even long polynomials can be evaluated in short latency.

Roughly speaking, latency grows logarithmically in the degree of polynomials. The permissive environment that allows for functions that return accurate 53-bit results should be contrasted with that which is required for functions that return accurate 64-bit results. Some functions returning accurate 64-bit results are provided in a special double-extended `libm` as well as in IA-32 compatibility operations [6]. In both, considerable effort was taken to minimize rounding error. Often, computations were carefully choreographed into a dominant part that was calculated exactly and a smaller part that was subject to rounding error. We frequently stored precomputed values in two pieces to maintain

intermediate accuracy beyond the underlying precision of 64 bits. All these costly implementation techniques are unnecessary in our present double-precision context.

We summarize the above as four simple principles:

1. Use a simple reduction scheme, even if such a scheme only works for a subset of the argument domain, provided this subset represents the most common situations.
2. Consider novel reduction methods that avoid memory latency.
3. Use tables of moderate size.
4. Do not fear long polynomials. Instead, work hard at using parallelism to minimize latency.

In the next sections, we show these four principles in action on Merced, the first implementation of the IA-64 architecture.

SIMPLE AND FAST RANGE REDUCTION

A common reduction step involves the calculation of the form

$$r = x - N\,\mathbf{r}.$$

This includes the forward trigonometric functions sin, cos, tan, and the exponential function exp, where $\mathbf{r}$ is of the form $\mathbf{p}/2^K$ for the trigonometric functions and of the form $\ln(2)/2^K$ for the exponential function. We exploit the fact that the overwhelming majority of arguments will be in a limited range. For example, the evaluation of trigonometric functions like sin for very large arguments is known to be costly. This is because to perform a range reduction accurately by subtracting a multiple of $\mathbf{p}/2^K$, we need to implicitly have a huge number of bits of $\mathbf{p}$ available. But for inputs of less than 2^{10} in magnitude, the reduction can be performed accurately and efficiently. The overwhelming majority of cases will fall within this limited range. Other more time-consuming procedures are well known and are required when arguments exceed 2^{10} in magnitude (see [3] and [6]).

The general difficulty of range reduction implementation is that $\mathbf{r}$ is not a machine number. If we compute:

$$r = x - N\,P$$

where the machine number P approximates $\mathbf{p}/2^K$, then if x is close to a root of the specific trigonometric function, the small error, $\mathbf{e} = |P - \mathbf{p}/2^K|$, scaled up by N, constitutes a large relative error in the final result. However, by using number-theoretic arguments, one can see that when reduction is really required for double-precision numbers in the specified range, the result of any of the trigonometric functions, sin, cos, and tan, cannot be smaller in magnitude than about 2^{-60} (see [7]).

The worst relative error (which occurs when the result of the trigonometric function is its smallest, $2^{60,}$ and N is close to 2^{10+K}) is about $2^{70+K}\,\varepsilon$. If we store P as two double-extended precision numbers, $P\_1 + P\_2$, then we can make $\mathbf{e} < 2^{-130-K}$ sufficient to make the relative error in the final result negligible.

One technique to provide an accurate reduced argument on IA-64 is to apply two successive `fma` operations

$$r_0 = x - N\,P\_1; \quad r = r_0 - N\,P\_2.$$

The first operation introduces no rounding error because of the well known phenomenon of cancellation.

For sin and cos, we pick K to be 4, so the reconstruction has the form

$$\sin(x) = \sin(N\mathbf{p}/16)\cos(r) + \cos(N\mathbf{p}/16)\sin(r)$$

and

$$\cos(x) = \cos(N\mathbf{p}/16)\cos(r) - \sin(N\mathbf{p}/16)\sin(r).$$

Periodicity implies that we need only tabulate $\sin(N\mathbf{p}/16)$ and $\cos(N\mathbf{p}/16)$ for $N = 0, 1, \ldots, 31$.

The case for the exponential function is similar. Here $\ln(2)/2^K$ (K is chosen to be 7 in this case) is approximated by two machine numbers $P\_1 + P\_2$, and the argument is reduced in a similar fashion.

NOVEL REDUCTION

Some mathematical functions f have the property that

$$f(u\,v) = g(f(u), f(v))$$

where g is a simple function such as the sum or product operator. For example, for the logarithm, we have (for positive u and v)

$$\ln(u\,v) = \ln(u) + \ln(v) \quad (g \text{ is the sum operator})$$

while for the cube root, we have

$$(u\,v)^{1/3} = u^{1/3}\,v^{1/3} \quad (g \text{ is the product operator}).$$

In such situations, we can perform an argument reduction very quickly using IA-64's basic floating-point reciprocal approximation (`frcpa`) instruction, which is primarily intended to support floating-point division. According to its definition, `frcpa`(a) is a floating-point with 11 significant bits that approximates $1/a$ using a lookup on the top 8 bits of the (normalized) input number a. This 11-bit floating-point number approximates $1/a$ to about 8 significant bits of accuracy. The exact values returned are specified in the IA-64 architecture definition. By enumeration of the approximate reciprocal values, one can show that for all input values a,

$$\texttt{frcpa}(a) = (1/a)\,(1 - \mathbf{b}), \quad |\mathbf{b}| \le 2^{-8.86}.$$

We can write $f(x)$ as

$$f(x) = f(x \cdot \texttt{frcpa}(x) / \texttt{frcpa}(x))$$
$$= g(\ f(x \cdot \texttt{frcpa}(x)),\ f(1/\texttt{frcpa}(x))\ ).$$

The $f(1/\texttt{frcpa}(x))$ terms can be stored in precomputed tables, and they can be obtained by an index based on the top 8 bits of x (which uniquely identifies the corresponding frcpa (x)).

Because the *f's* we are considering here have a natural expansion around 1,

$$f(x \cdot \texttt{frcpa}(x))$$

is most naturally approximated by a polynomial evaluated at the argument $r = x \cdot \texttt{frcpa}(x) - 1$. Hence, a single fma constitutes our argument reduction computation, and the value frcpa(x) is obtained without any memory latency.

We apply this strategy to $f(x) = \ln(x)$.

$$\ln(x) = \ln(1/\texttt{frcpa}(x)) + \ln(\texttt{frcpa}(x)\, x)$$
$$= \ln(1/\texttt{frcpa}(x)) + \ln(1 + r)$$

The first value on the right-hand side is obtained from a table, and the second value is computed by a minimax polynomial approximating $\ln(1+r)$ on $|r| \le 2^{-8.8}$. The quantity $2^{-8.8}$ is characteristic of the accuracy of the IA-64 frcpa instruction.

The case for the cube root function cbrt is similar.

$$(x)^{1/3} = (1/\texttt{frcpa}(x))^{1/3} (\texttt{frcpa}(x)\, x)^{1/3}$$
$$= (1/\texttt{frcpa}(x))^{1/3} (1 + r)^{1/3}.$$

The first value on the right-hand side is obtained from a table, and the second value is computed by a minimax polynomial approximating $(1+r)^{1/3}$ on $|r| \le 2^{-8.8}$.

MODERATE TABLE SIZES

We tabulate here the number of double-extended table entries used in each function. The trigonometric functions sin and cos share the same table, and the functions tan and atan do not use a table at all.

Function	Number of Double-Extended Entries
cbrt	256 (3072 bytes)
exp	24 (288 bytes)
Ln	256 (3072 bytes)
sin, cos	64 (768 bytes)
tan	None
atan	None

Table 1: Table sizes used in the algorithms

Table 1 does not include the number of constants for argument reduction nor does it include the number of coefficients needed for evaluating the polynomial.

OPTIMAL EVALUATION OF POLYNOMIALS

The traditional Horner's rule of evaluation of a polynomial is efficient on serial machines. Nevertheless, a general degree-n polynomial requires a latency of n fma's. When more parallelism is available, it is possible to be more efficient by splitting the polynomial into parts, evaluating the parts in parallel, and then combining them. We employ this technique to the polynomial approximation steps for all the functions. The enhanced performance is crucial to the cases of tan and atan where the polynomials involved are of degrees 15 and 22. Even for the other functions where the polynomials are varying in degree from 4 to 8, our technique also contributes to a noticeable gain over the straightforward Horner's method. We now describe this technique in more detail.

Merced has two floating-point execution units, so there is certainly some parallelism to be exploited. Even more important, both floating-point units are fully pipelined in five stages. Thus, two new operations can be issued *every* cycle, even though the results are then not available for a further five cycles. This gives much of the same benefit as more parallel execution units. Therefore, as noted by the author in reference [3], one can use more sophisticated techniques for polynomial evaluation intended for highly parallel machines. For example, Estrin's method [2] breaks the evaluation down into a balanced binary tree.

We can easily place a lower bound on the latency with which a polynomial can be computed: if we start with x and the coefficients c_i, then by induction, in n serial fma operations, we cannot create a polynomial that is a degree higher than 2^n, and we can only equal 2^n if the term of the highest degree is simply x^{2^n} with unity as its coefficient. For example, in one operation we can reach $c_0 + c_1 x$ or $x + x^2$ but not $x + c_0 x^2$. Our goal is to find an actual scheduling that comes as close as possible to this lower bound.

Simple heuristics based on binary chopping normally give a good evaluation strategy, but it is not always easy to visualize all the possibilities. When the polynomial can be split asymmetrically, or where certain coefficients are special, such as 1 or 0, there are often ways of doing slightly better than one might expect in overall latency or at least in the number of instructions required to attain that latency (and hence in throughput). Besides, doing the scheduling by hand is tedious. We search automatically for the best scheduling using a program that exhaustively examines all essentially different scheduling. One simply

enters a polynomial, and the program returns the best latency and throughput attainable, and it lists the main ways of scheduling the operations to attain this.

Even with various intelligent pruning approaches and heuristics, the search space is large. We restrict it somewhat by considering only `fma` combinations of the form $p_1(x) + x^k p_2(x)$. That is, we do not consider multiplying two polynomials with nontrivial coefficients. Effectively, we allow only solutions that work for arbitrary coefficients, without considering special factorization properties. However, for polynomials where all the coefficients are 1, these results may not be optimal because of the availability of nontrivial factorizations that we have ruled out. For example, we can calculate:

$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6$$

as

$$1 + (1 + (x + x^2))\,(x + (x^2)\,(x^2))$$

which can be scheduled in 15 cycles. However, if the restriction on `fma` operations is observed, then 16 cycles is the best attainable.

The optimization program works in two stages. First, all possible evaluation orders using these restricted `fma` operations are computed. These evaluation orders ignore scheduling, being just "abstract syntax" tree structures indicating the dependencies of subexpressions, with interior nodes representing `fma` operations of the form $p_1(x) + x^k\, p_2(x)$:

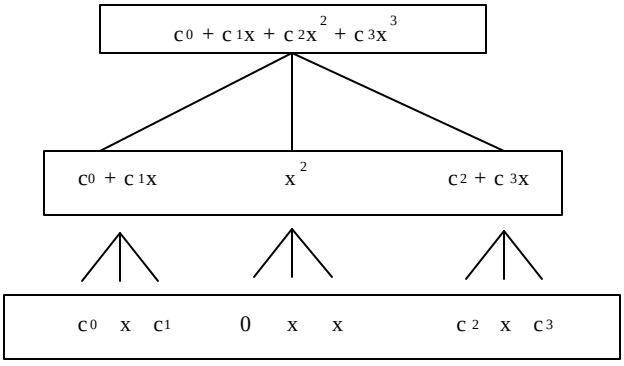

Figure 1: A dependency tree

However, because of the enormous explosion in possibilities, we limit the search to the smallest possible tree depth. This tree depth corresponds to the minimum number of serial operations than can possibly be used to evaluate the expression using the order denoted by that particular tree. Consequently, if the tree depth is d then we cannot possibly do better than $5d$ cycles for that particular tree. Now, assuming that we can in fact do at least as well as $5d + 4$, we are justified in ignoring trees of a depth greater than or equal to $d + 1$, which could not possibly be scheduled in as few cycles. This turns out to be the case for all our examples.

The next stage is to take each tree (in some of the examples below there are as many as 10000 of them) and calculate the optimal scheduling. The optimal scheduling is computed backwards by a fairly naive greedy algorithm, but with a few simple refinements based on stratifying the nodes from the top as well as from the bottom.

The following table gives the evaluation strategy found by the program for the polynomial:

$$x + c_2x^2 + c_3x^3 + \ldots + c_9x^9$$

Table 2 shows that it can be scheduled in 20 cycles, and we have attained the lower bound. However, if the first term were c_1x we would need 21.

Cycle	FMA Unit 1	FMA Unit 2
0	$v_1 = c_2 + x\ c_3$	$v_2 = x\ x$
3	$v_3 = c_6 + x\ c_7$	$v_4 = c_8 + x\ c_9$
4	$v_5 = c_4 + x\ c_5$	
5	$v_6 = x + v_2\ v_1$	$v_7 = v_2\ v_2$
9	$v_8 = v_3 + v_2\ v_4$	
10	$v_9 = v_6 + v_7\ v_5$	$v_{10} = v_2\ v_7$
15	$v_{11} = v_9 + v_{10}\ v_8$	

Table 2: An optimal scheduling

OUTLINE OF ALGORITHMS

We outline each of the seven algorithms discussed here. We concentrate only on the numeric cases and ignore situations such as when the input is out of the range of the functions' domains or non-numeric (NaN for example).

Cbrt

1. *Reduction:* Given x, compute $r = x$ `frcpa`$(x) - 1$.
2. *Approximation*: Compute a polynomial p(r) of the form $p(r)=p_1r + p_2r^2 + \ldots + p_6r^6$ that approximates $(1+r)^{1/3}-1$.
3. *Reconstruction:* Compute the result $T + Tp(r)$ where the T is $(1/$`frcpa`$(x))^{1/3}$. This value T is obtained via a tabulation of $(1/$`frcpa`$(y))^{1/3}$, where $y=1+k/256$, k ranges from 0 to 255 and a tabulation of $2^{-j/3}$, and j ranges from 0 to 2.

Exp

1. *Reduction:* Given x, compute N, the closest integer to the value $x\ (128/\ln(2))$. Then compute $r = (x - N\ P_1) - N\ P_2$. Here P_1+P_2 approximates $\ln(2)/128$ (see previous discussions).

2. *Approximation:* Compute a polynomial $p(r)$ of the form $p(r) = r + p_1 r^2 + \ldots + p_4 r^5$ that approximates $\exp(r) - 1$.

3. *Reconstruction:* Compute the result $T + Tp(r)$ where T is $2^{N/128}$. This value T is obtained as follows. First, N is expressed as $N = 128\,M + 16\,K + J$, where I_1 ranges from 0 to 15, and I_2 ranges from 0 to 7. Clearly $2^{N/128} = 2^M\ 2^{K/8}\ 2^{J/128}$. The first of the three factors can be obtained by scaling the exponent; the remaining two factors are fetched from tables with 8 entries and 16 entries, respectively.

Ln

1. *Reduction:* Given x, compute $r = x \cdot \texttt{frcpa}(x) - 1$.

2. *Approximation:* Compute a polynomial $p(r)$ of the form $p(r) = p_1 r^2 + \ldots + p_5 r^6$ that approximates $\ln(1+r) - r$.

3. *Reconstruction:* Compute the result $T + r + p(r)$ where the T is $\ln(1/\texttt{frcpa}(x))$. This value T is obtained via a tabulation of $\ln(1/\texttt{frcpa}(y))$, where $y = 1 + k/256$, k ranges from 0 to 255, and a calculation of the form $N \ln(2)$.

Sin and Cos

We first consider the case of $\sin(x)$.

1. *Reduction:* Given x, compute N, the closest integer to the value $x \cdot (16/\pi)$. Then compute $r = (x - N \cdot P_1) - N \cdot P_2$. Here $P_1 + P_2$ approximates $\pi/16$ (see previous discussions).

2. *Approximation:* Compute two polynomials: $p(r)$ of the form $r + p_1 r^3 + \ldots + p_4 r^9$ that approximates $\sin(r)$ and $q(r)$ of the form $q_1 r^2 + q_2 r^4 + \ldots + q_4 r^8$ that approximates $\cos(r) - 1$.

3. *Reconstruction:* Return the result as $Cp(r) + (S + Sq(r))$ where C is $\cos(N \cdot \pi/16)$ and S is $\sin(N \cdot \pi/16)$ obtained from a table.

The case of $\cos(x)$ is almost identical. Add 8 to N just after it is first obtained. This works because of the identity $\cos(x) = \sin(x + \pi/2)$.

Tan

1. *Reduction:* Given x, compute N, the closest integer to the value $x \cdot (2/\pi)$. Then compute $r = (x - N \cdot P_1) - N \cdot P_2$. Here $P_1 + P_2$ approximates $\pi/2$ (see previous discussions).

2. *Approximation:* When N is even, compute a polynomial $p(r) = r + r\,t\,(p_0 + p_1 t + \ldots + p_{15} t^{15})$ that approximates $\tan(r)$. When N is odd, compute a polynomial $q(r) = (-r)^{-1} + r(q_0 + q_1 t + \ldots + q_{10} t^{10})$ that approximates $-\cot(r)$. The term t is r^2. We emphasize the fact that parallelism is fully utilized.

3. *Reconstruction:* If N is even, return p. If N is odd, return q.

Atan

1. *Reduction:* No reduction is needed.

2. *Approximation:* If $|x|$ is less than 1, compute a polynomial $p(x) = x + x^3(p_0 + p_1 y + \ldots + p_{22} y^{22})$ that approximates $\text{atan}(x)$, y is x^2. If $|x| > 1$, compute several quantities, fully utilizing parallelism. First, compute $q(x) = q_0 + q_1 y + \ldots + q_{22} y^{22}$, $y = x^2$, that approximates $x^{45} \text{atan}(1/x)$. Second, compute c^{45} where $c = \texttt{frcpa}(x)$. Third, compute another polynomial $r(\beta) = 1 + r_1 \beta + \ldots + r_{10} \beta^{10}$, where β is the quantity $x \cdot \texttt{frcpa}(x) - 1$ and $r(\beta)$ approximates the value $(1 - \beta)^{-45}$.

3. *Reconstruction:* If $|x|$ is less than 1, return $p(x)$. Otherwise, return $\text{sign}(x)\pi/2 - c^{45} r(\beta) q(x)$. This is due to the identity $\text{atan}(x) = \text{sign}(x)\pi/2 - \text{atan}(1/x)$.

SPEED AND ACCURACY

These new double-precision elementary functions are designed to be both fast and accurate. We present the speed of the functions in terms of latency for arguments that fall through the implementation in a path that is deemed most likely. As far as accuracy is concerned, we report the largest observed error after extensive testing in terms of units of last place (ulps). This error measure is standard in this field. Let f be the mathematical function to be implemented and F be the actual implementation in double precision. When $2^L < |f(x)| \le 2^{L+1}$, the error in ulps is defined as $|f(x) - F(x)| / (2^{L-52})$. Note that the smallest worst-case error that one can possibly attain is *0.5* ulps. Table 3 tabulates the latency and maximum error observed.

Function	Latency (cycles)	Max. Error (ulps)
cbrt	60	0.51
exp	60	0.51
ln	52	0.53
sin	70	0.51
cos	70	0.51
tan	72	0.51
atan	66	0.51

Table 3: Speed and accuracy of functions

CONCLUSIONS

We have shown how certain key features of the IA-64 architecture can be exploited to design transcendental functions featuring an excellent combination of speed and accuracy. All of these functions performed over twice as fast as the ones based on the simple conversion of a library tailored for double-extended precision. In one instance, the ln function described here contributed to a two point increment of SpecFp benchmark run under simulation.

The features of the IA-64 architecture that are exploited include parallelism and the fused multiply add as well as less obvious features such as the reciprocal approximation instruction. When abundant resources for parallelism are available, it is not always easy to visualize how to take full advantage of them. We have searched for optimal instruction schedules. Although our search method is sufficient to handle the situations we have faced so far, more sophisticated techniques are needed to handle more complex situations. First, polynomials of a higher degree may be needed in more advanced algorithms. Second, more general expressions that can be considered as multivariate polynomials are also anticipated. Finally, our current method does not handle the full generality of microarchitectural constraints, which also vary in future implementations on the IA-64 roadmap. We believe this optimal scheduling problem to be important not only because it yields high-performance implementation, but also because it may offer a quantitative analysis on the balance of microarchitectural parameters. Currently we are considering an integer programming framework to tackle this problem. We welcome other suggestions as well.

REFERENCES

[1] Cody Jr., William J. and Waite, William, *Software Manual for the Elementary Functions*, Prentice Hall, 1980.

[2] Knuth, D.E., *The Art of Computer Programming vol. 2: Seminumerical Algorithms*, Addison-Welsey, 1969.

[3] Muller, J. M., *Elementary functions: algorithms and implementation*, Birkhaüser, 1997.

[4] Payne, M., "An Argument Reduction Scheme on the DEC VAX," *Signum Newsletter*, January 1983.

[5] Powell, M.J.D., *Approximation Theory and Methods*, Cambridge University Press, 1981.

[6] Story, S. and Tang, P.T.P., "New algorithms for improved transcendental functions on IA-64," in *Proceedings of 14th IEEE symposium on computer arithmetic*, IEEE Computer Society Press, 1999.

[7] Smith, Roger A., "A Continued-Fraction Analysis of Trigonometric Argument Reduction," *IEEE Transactions on Computers*, pp. 1348-1351, Vol. 44, No. 11, November 1995.

[8] Tang, P.T.P., "Table-driven implementation of the exponential function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 15, pp. 144-157, 1989.

AUTHORS' BIOGRAPHIES

John Harrison has been with Intel for just over one year. He obtained his Ph.D. degree from Cambridge University in England and is a specialist in formal validation and theorem proving. His e-mail is johnh@ichips.intel.com.

Ted Kubaska is a senior software engineer with Intel Corporation in Hillsboro, Oregon. He has a M.S. degree in physics from the University of Maine at Orono and a M.S. degree in computer science from the Oregon Graduate Institute. He works in the MSL Numerics Group where he implements and tests floating-point algorithms. His e-mail is Theodore.E.Kubaska@intel.com.

Shane Story has worked on numerical and floating-point related issues since he began working for Intel eight years ago. His e-mail is Shane.Story@intel.com.

Ping Tak Peter Tang (his friends call him Peter) joined Intel very recently as an applied mathematician working in the Computational Software Lab of MSL. Peter received his Ph.D. degree in mathematics from the University of California at Berkeley. His interest is in floating-point issues as well as fast and accurate numerical computation methods. Peter has consulted for Intel in the past on such issues as the design of the transcendental algorithms on the Pentium, and he contributed a software solution to the Pentium division problem. His e-mail is Peter.Tang@intel.com.