View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Comparing WCET and Resource Demands of Trigonometric Functions Implemented as Iterative Calculations vs. Table-Lookup *

Raimund Kirner, Markus Grössing, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien, Austria
raimund@vmars.tuwien.ac.at

Abstract

Trigonometric functions are often needed in embedded real-time software. To fulfill concrete resource demands, different implementation strategies of trigonometric functions are possible.

In this paper we analyze the resource demands of iterative calculations compared to other implementation strategies, using the trigonometric functions as a case study. By analyzing the worst-case execution time (WCET) of the different calculation techniques of trigonometric functions we got the surprising result that the WCET of iterative calculations is quite competitive to alternative calculation techniques, while their economics on memory demand is far superior. Finally, a discussion of the general applicability of the obtained results is given as a design guide for embedded software.

1 Introduction

For real-time systems in safety-critical environments it is indispensable to design the temporal behavior of the system based on knowledge of the worst-case execution time (WCET) of the real-time tasks. A general discussion on research directions in the area of WCET analysis can be found in [9].

Besides analyzing the timing behavior of programs we also look at software design techniques that proactively simplify the analysis of the WCET. We have described a general paradigm, which we call WCET-oriented programming [10]. The basic idea of WCET-oriented programming can be summarized as the search for algorithms whose execution-time variability is small, for example, by avoiding input-data dependent control flow decisions whenever possible.

In this paper we study the characteristics of iteration-based computation techniques, we address interesting questions like whether these algorithms are suitable for real-time computing. For example, it is a common belief that iteration-based computation is critical, because a) long execution times due to high number of needed iterations and b) the problem of finding a precise upper iteration bound.

In the here-presented case study, we look at the behavior of trigonometric functions. The contribution of this paper is to connect the known properties of trigonometric functions to implementation techniques of real-time software and to provide an analysis of relevant characteristics like computation time and memory demands. Besides the interesting results obtained from our analysis, we also describe the application of WCET analysis to embedded software with floating point emulation.

2 Related Work

The work in this paper focuses on the properties of time-memory tradeoffs for real-time software. For example, one might design algorithms with shorter execution time by using more memory.

Sorting examples are *sorting by counting*, where a second array is used to sort elements with known relative positions based on their key in linear time, and *Radix Sort* [6]. Alternatively, *lookup tables* (LUT) can be used to reduce online calculations by deploying precalculated values. As an example for the use of lookup tables, see [7]. Time-space tradeoffs on dictionary attacks to break passwords are presented in [8]. Three further examples of applying time-memory tradeoffs are described in [11].

*This work has been partially supported by the FIT-IT research project "Model-Based Development of distributed Embedded Control Systems (MoDECS)" and the ARTIST2 Network of Excellence of IST FP6.

ECRTS 2006
6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/669

3 Trigonometric Functions

For our study of different computation techniques we focus on trigonometric functions because they are heavily used in many scientific disciplines. *Sine*, *cosine* and *tangent* as well as their inverse functions play important roles not only in surveying, navigation, or scientific mathematics, but also in many other fields like acoustics, astronomy, computer graphics, electrical engineering and electronics, mechanical engineering, optics, etc.

First of all, it is important to keep in mind that the requirements on trigonometric functions are quite different depending on the application domain. For example, an application domain where performance is typically more important than precision is 3D computer graphics. The cosine is a fundamental operation in 3D rendering techniques like various shading methods, ray tracing, etc. [12]. Those rendering techniques have to use the trigonometric functions excessively often. Therefore, effective approximation techniques are very important to gain performance, while high precision is not a first-order requirement.

However, we are focusing more on the use of trigonometric functions in the domain of embedded real-time systems. They are used in mechanical applications, e.g., to determine distances in automation systems, or for controlling the movement of a robotic arm. Other important real-time applications are multimedia systems, where they are used to compute Fourier transforms (e.g., for audio processing) or discrete cosine transforms (for graphics) are performed. Further interesting application fields are applications that use ultrasound, optical devices, or statistical computations.

For the application of trigonometric functions in embedded real-time control systems, typically both, the numerical precision and the resource demands are relevant. For the following discussions of different calculation techniques we concentrate on the *cosine* function, since the other trigonometric functions are closely related respectively can be derived from it. We also discuss the maximum error for each calculation technique, which is needed for the comparison of iterative calculation and table-lookup in Section 4.

3.1 Iterative Approximation (Taylor Series)

In common implementations of trigonometric functions the Taylor series is used to approximate sine, cosine and tangent. There also exist other iterative algorithms like CORDIC [1], which are slower than Taylor series but easier to implement in hardware as it does not need multiplication operations. In this paper we focus on the Taylor series because we are interested in implementations in software. Let us consider the power series implementation of cosine (Equation 1): to reach full *double* precision (as defined in [3]) a Taylor polynomial of degree 14 is needed. As only the coefficients of even powers are significant to calculate the cosine function, only seven coefficients are needed. We call this class of cosine implementation techniques CTAYLOR.

$$\cos(x) = \sum_{n=0}^{\infty}(-1)^n \cdot \frac{x^{2n}}{(2n)!} \approx \sum_{n=0}^{7}(-1)^n \cdot \frac{x^{2n}}{(2n)!} \quad (1)$$

The constant coefficients do not need to be calculated at runtime everytime the function is called. Instead, they can be stored as static constants.

The accuracy of the power series decreases as the distance of the argument from the center grows. Therefore for trigonometric functions this distance is limited to $\pi/4$. As the center of sine, cosine, and tangent approximation is chosen to be zero the actual evaluation interval of these functions is $[-\pi/4; \pi/4]$. To evaluate arguments outside this interval an argument reduction needs to be performed [7].

To estimate the maximum error of a Taylor series implementation we need to consider the error at $\pi/4$, where the distance to the center of the power series is maximal.

The maximum error of Taylor series with n iterations is given by the remainder term R_{n+1} in Equation 2.

$$R_{n+1} = \frac{1}{n!}\int_0^x (x-t)^n \cdot \cos^{(n+1)}(t)dt =$$
$$= \cos(x) - \left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} \cdots + (-1)^n \cdot \frac{x^{2n}}{2n!}\right) \quad (2)$$

3.2 Approximation using Lookup Tables

Lookup tables are commonly used to replace runtime calculations with simpler lookup operations. Retrieving an array value from memory is usually much faster than making an expensive computation.

In the following we will take a look on three different implementations of lookup tables:

- Fast and simple lookup tables (*FLUT*)
- Equidistantly interpolated lookup tables (*EDI-LUT*), and
- Lookup table with interpolation with smart placement of interpolation points (*SMILUT*)

3.2.1 Fast and Simple Table Lookup

The fast and simple lookup table (FLUT) is nothing more than a data array that stores pre-calculated function values of the function in it. The places where these values are taken are equidistant, so the array index fitting to a given argument can easily be computed. Each value in the array covers an interval of arguments. The biggest error occurs, where the function has its greatest gradient (see Equation 3. For example, a cosine lookup table in the interval $[0; \pi/2]$ has its greatest gradient at $\pi/2$.

$$E_{max}(n) = \cos\left(\frac{\pi}{2} - \frac{\pi}{4(n-1)}\right) = \sin\left(\frac{\pi}{4(n-1)}\right) \quad (3)$$

The advantage of FLUT is that it is easy to implement and the estimation of the timing behavior is simple. The performance according to speed is very good but accuracy requirements should not be too high. To enhance accuracy or to reduce the table size if a particular level of accuracy is given other methods like EDILUT and SMILUT can be used.

3.2.2 Table Lookup with Equidistant Interpolation

An equidistant interpolated lookup table (EDILUT) reaches significantly higher accuracy compared to a FLUT of the same size. The price to pay is a little more arithmetics and so longer execution time.

The entries of an EDILUT are the function values of equidistantly distributed places of the input interval. In the case of a *cosine* EDILUT the input interval is $[0; \pi/2]$. In a cosine calculation, first the two interpolation points next to the given argument are determined. Then a straight line through these two points is calculated and the argument is set into this straight interpolation line. With this method accuracy can be increased significantly.

The maximum error of EDILUT occurs not on the place with the greatest gradient, like it was the case for FLUT, but on the place with the greatest curvature. For our *cosine* function this is the case near the origin, so we expect the greatest error to occur in the first interpolation interval. The error of the interpolation in the first interval can be calculated by subtracting the linear interpolation between the first two interpolation points from the function. By deriving this function and setting to zero, the exact place of the maximum error is retrieved. Applying this value to the error function gives the maximum absolute error (Equation 4) for a concrete lookup table size n of EDILUT.

$$E_{max}(n) = \cos\left(\sin^{-1}(k)\right) - k \cdot \sin^{-1}(k) - 1, \quad k = -\frac{1 - \cos\frac{\pi}{2(n-1)}}{\frac{\pi}{2(n-1)}} \quad (4)$$

3.2.3 Table Lookup with Smart Interpolation

A smart interpolated lookup table (SMILUT) is a further improvement of EDILUT. In a SMILUT the interpolation points are not equidistantly distributed in the input interval but in a smarter way. The function for mapping the input interval into the range of array indices should be rather simple. We map the input interval to the indices using the squareroot function. The result is an improvement of accuracy.

With this placement we achieve that the maximum error does not occur within the first interpolation interval but rather in the middle of the overall input interval.

As the squareroot function might be too expensive to compute, we considered an alternative implementation for finding the correct interpolation interval, namely using binary search.

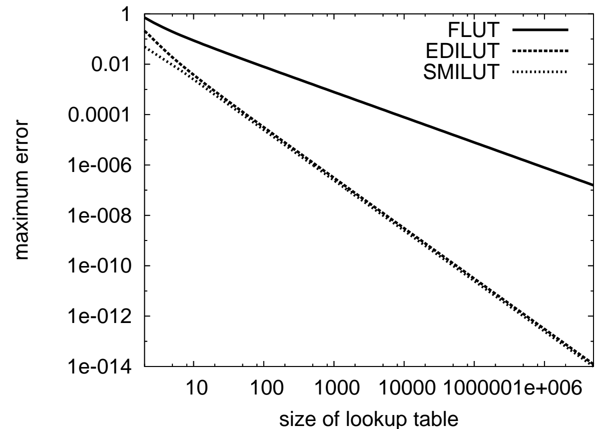

Figure 1. Maximum Absolute Error of FLUT, EDILUT, and SMILUT

To complete the discussion about LUT-based solutions, a comparison of the accuracy in dependence of the size of the lookup array is given in Figure 1 [1].

[1] Note that due to range limitations in the numerical calculation, the size values above 2000 are extrapolated values to show the tendency of the graph.

4 Comparison of Iterative vs. LUT-based Techniques

A comparison of different iteration numbers of Taylor series implementations to the three lookup table approaches is depicted in Figure 2. It is shown how many entries a particular type of lookup table needs to exceed the accuracy of different Taylor series implementations.

As the maximum error of FLUT and EDILUT can be calculated analytically, these two variants can be easily compared to the Taylor series. For the comparison of SMILUT a simple tool was developed to experimentally determine the required size of the SMILUT to reach the accuracy of the different Taylor series implementations. This tool determines the maximum error of a SMILUT for a given array size. If the error is too big the array size is increased. The program terminates when the accuracy of the SMILUT exceeds a given limit, e.g., the accuracy of a particular Taylor implementation.

As shown in Figure 2 the accuracy of FLUT is much worse than the accuracy of the other LUT implementations. SMILUT performs slightly better than EDILUT. One can see that the size of lookup tables of any type grows exponentially with the number of Taylor iterations. Thus, if high accuracy requirements need to be met, the use of lookup tables may not be feasible or sensible to approximate trigonometric functions - the memory consumption of these algorithms is too high.

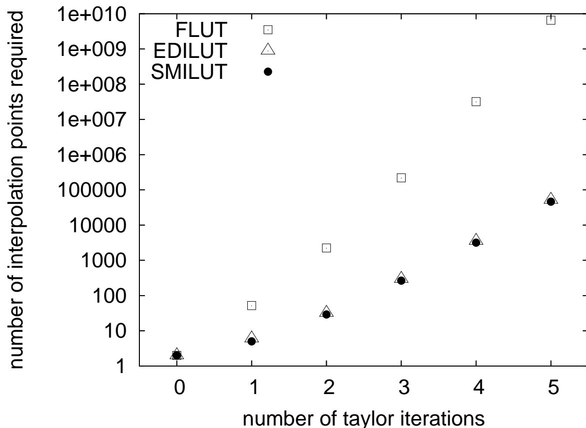

Figure 2. Necessary LUT Size to Match the Accuracy of Taylor Series

5 Experimental Evaluation

In Section 3 we described the theoretical properties of different calculation techniques of trigonometric functions. Lets now look at the different calculation techniques from a practical point of view. Especially interesting for the use of trigonometric functions in embedded real-time systems are their resource demands. Therefore, we analyzed their memory footprints in data and code memory, and calculated an upper bound of their worst-case execution time (WCET).

5.1 Studied Algorithms

To analyze the properties of the different computation techniques discussed in Section 3 on a concrete computer platform, we implemented several variants of the cosine function. We implemented the cosine function for the *double* data type of ANSI C (which is typically the 64-bit IEEE floating point format [3]).

Two iterative cosine variants belonging to CTAYLOR were implemented, one which is a straight forward implementation of the Taylor-formula and one with precalculation of the coefficients of the Taylor-terms.

On the other side the three LUT-based variants of Section 3.2 have been implemented: FLUT the straight forward method, EDILUT which uses linear interpolation and SMILUT, an implementation using binary search to find the correct interpolation point within the LUT. Compared to EDILUT it is highly performance oriented and uses more precomputed results, requiring three LUTs.

Some characteristic parameters of the different cosine implementations are given in Table 1. The column *#BB* denotes the number of *basic blocks* of the generated object code. The *DataMem* columns give the required number of bytes to store the intermediate data and the LUT. It is given first in parametric form as a function of the LUT size N, and second for the concrete case $N = 1000$. The column *CodeMem* denotes the net code size, i.e., without counting the standard library functions which are linked by the compiler. The byte values are given for the Infineon C167 processor, a 16bit architecture.

5.2 WCET Analysis

In the following we describe how we derived the WCET of the different cosine implementation techniques. Our WCET analysis tool *calc_wcet_167* [2] uses static timing analysis to calculate an upper bound of a task's WCET. The target architecture for the tool is the processor C167 from Infineon, for which the GCC compiler was ported by the company *HighTec EDV*

[2] `http://www.wcet.at/tools.html`

Function name	*#BB*	*DataMem* (parametric) [bytes]	(N=1000) [bytes]	*CodeMem* [bytes]
CTAYLOR	22	28	n.a.	720
CTAYLOR_tab	19	90	n.a.	598
FLUT	13	10+N·8	8 010	456
EDILUT	15	34+N·8	8 034	902
SMILUT	23	30+(N+1)·24	24 054	536

Table 1. Implemented Calculation Variants of the Cosine Function

Systeme GmbH [3]. The integration of optimizing compilation into the WCET analysis is described in [5]. The development and verification of the timing model for the Infineon C167 is documented in [2]. Because the Infineon C167 processor has a relatively simple architecture, the overestimation of the calculated WCET bound of our tool is tight, maximal 5%, but typically less than 2%, provided the control flow is precisely modelled by flow constraints [2].

The cosine implementations were written in WCETC, based on a subset of ANSI C but providing additional features to annotate the source code with flow information to guide the WCET analysis tool [4].

The WCET analysis of the cosine implementations itself did not require anything special to mention. However, the overall WCET analysis was not easy because the Infineon C167 processor does not have a floating point engine in hardware. For such architectures the compiler links extra program code that emulates the floating point computations in software (`libsgnu.a` provided by *HighTec*). To perform the WCET analysis we disassembled the object code of the library and annotated it at assembly code level with flow information.

The final WCET analysis results of the different cosine implementations are given in Table 2. Besides the properties of the concrete implementations, these values are generally rather high because we assumed a slow hardware configuration with slow external memory. The first of the WCET columns shows the WCET in a parametric form. This is only relevant for the iterative implementations, where *iter* is the number of loop iterations used to iteratively refine the result based on the Taylor series of the cosine function. The other four WCET columns show the WCET bound of the iterative algorithms for different iteration counts.

5.3 Discussion

The results of the general analysis of the maximum absolute error for CTAYLOR and of the maximum absolute error of FLUT, EDILUT, and SMILUT together with the precision relationsship between CTAYLOR and LUT-based methods (Figure 2) can be combined with the results from the concrete implementation to reason about the pros and cons of the different computation paradigms.

To demonstrate how this can be done, lets assume that for a concrete project one needs a cosine function providing a maximal error of less than $2.5 \cdot 10^{-8}$. Evaluating Equation 2 at $\pi/4$ it follows that one would need 4 iterations ($\equiv$ 5 Taylor terms) with the CTAYLOR methods. To replace the CTAYLOR method later by an adequate LUT-based method one could deduce from Figure 2 the required LUT size to match at least the same precision. For example, to obtain the same quality with EDILUT or SMILUT, one has to choose an LUT size of $N > 3000$!

From Table 1 we can see that in this case the additional memory demand for the LUT-based methods is significant. The FLUT method, though it is relatively fast, is completely out of choice as it would require an LUT size of $N > 10^5$. If performance really is the most important issue, then according to Table 2 one has to use the SMILUT implementation. But surprisingly, the CTAYLOR methods are not that bad regarding the WCET compared to SMILUT. As a rough indicator using our example, one would need only `90` bytes data memory when using CTAYLOR_tab compared to the more than `72kB` when using SMILUT. The code size is almost the same between these two implementations.

Another benefit of the CTAYLOR methods is their *anytime* characteristic. In case a CTAYLOR method gets interrupted, there is still some accuracy of the result available, for example, to move a robot arm at least in the intended direction, hoping that the control will be refined in the next round by a more accurate result. Depending on the application, this can be an advantage of iteration-based CTAYLOR methods compared to the LUT-based methods.

[3] `http://www.hightec-rt.com`

Function name	WCET [cycles] (parametric)	(iter=1)	(iter=3)	(iter=4)	(iter=7)
CTAYLOR	23 140 + iter·79 500	102 640	261 640	341 140	579 640
CTAYLOR_tab	23 380 + iter·49 200	72 580	170 980	220 180	367 780
FLUT	136 840	n.a.	n.a.	n.a.	n.a.
EDILUT	276 640	n.a.	n.a.	n.a.	n.a.
SMILUT	120 540	n.a.	n.a.	n.a.	n.a.

Table 2. Calculated WCET of the Cosine Functions (Target Processor: Infineon C167)

Generality of the Results

In our concrete case study of the cosine function it has been shown that the WCET of the iterative calculation is quite competitive to table-lookup while the economics on memory demand is far superior.

Iterative calculations generally provide the same advantages as long as they have a relatively compact calculation step within each iteration and the termination speed of the iterative calculation is reasonable.

As a further example, the *Newton method* to solve equations numerically tends to provide such a behavior, provided that the start value is already within the local convergence interval of the solution. There are many instantiations of the Newton method in practice, e.g., the *Heron method* to calculate the square root of a number.

6 Summary and Conclusion

Motivated by our general effort to study the suitability of different algorithms for real-time computing, we looked at different calculation techniques of trigonometric functions, because of their use in a wide range of technical applications.

One of the central conclusions is that whenever memory is highly constrained, iteration-based methods are very useful, because they tend to demand much less memory while still providing reasonable accuracy of results. And quite important, the performance overhead of iteration-based methods is not that high, even in our case study where we calculated the WCET for the C167, a processor that emulates floating point arithmetics in software. Further, the WCET analysis itself was an interesting experience, as we had to analyze routines of the floating-point emulation library by disassembling the object code.

In general, in embedded real-time systems, where size of memory is typically restricted, iterative algorithms can be a memory-efficient calculation technique without significant performance costs compared to LUT-based methods, as long as a reasonable termination speed of the iterative calculation is ensured.

References

[1] R. Andraka. A survey of cordic algorithms for fpga based computers. In *Proc. ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays*, pages 191–200, 1998.

[2] P. Atanassov. *Experimental Assessment of Worst-Case Program Execution Times*. PhD thesis, Technische Universität Wien, Vienna, May 2003.

[3] IEEE. *IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*. IEEE, New York, 1987. Reprinted in SIGPLAN Notices 22,2,9-25.

[4] R. Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[5] R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Vienna, Austria, May 2003.

[6] D. E. Knuth. *The Art of Computer Programming - Sorting and Searching*, volume 3. Addison Wesly, New York, USA, 2nd edition, 1998. ISBN 0-201-89685-0.

[7] J. N. Lygouras. Memory reduction in look-up tables for fast symmetric function generators. *IEEE Transactions on Instrumentation and Measurement*, 48(6):1254–1258, Dec. 1999.

[8] A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. 12th ACM conference on Computer and Communications Security*, pages 364–372, New York, NY, USA, 2005. ACM Press.

[9] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[10] P. Puschner and R. Kirner. Avoiding timing problems in real-time software. In *Proc. IEEE Computer Society's Workshop on Software Technologies for Future Embedded Systems*, May 2003.

[11] M. Stamp. Once upon a time-memory tradeoff. Technical report, San José State University, San José, California, USA, July 2003.

[12] A. Watt. *3D Computer Graphics*. Addison Wesley, 3rd edition, Dec. 1999. ISBN: 0201398559.