Measuring Time Cost of Built-in Functions

Mohamed Hefeeda

University of Connecticut

Computer Science and Engineering

Storrs, CT 06269

June 2000

This work was supported in part by the

Defense Advanced Research Projects Agency (DARPA)

Small Business Innovation Research (SBIR)

System Synthesis Environment Phase II
Measuring Time Cost of Built-in Functions

1 Introduction

Measuring the time cost of the built-in functions is a rather complicated task because there are so many factors that affect the time cost. The time cost is affected by –among other factors– the implementation of the built-in functions, number and type of arguments, the values of the arguments, and the operating environment.

This document presents two approaches for measuring the time cost of the built-in functions in the standard ANSI C language. The first one is a simple statistical approach with a reasonable accuracy, which InfoPike can implement easily and use it at least for the current version of HPMS. The second is a rather complicated analytical approach, which will give better general results in some sense. But it still needs some more research effort from UCONN and a considerable implementation effort from InfoPike to yield the desired results. Yet, we present this approach here for future extensions of HPMS.

The two approaches can be further extended in the future to better model the time cost of the built-in functions in different execution environments. For example, we can model the effect of the cache memory on the time cost. Clearly, the time cost will vary depending on where the data reside, in the cache or in the main memory.
2 Statistical approach

This is a simple straightforward statistical approach in which, we repeat measuring the time cost of a built-in function for many times and then calculate the mean and the variance. We should note that the time cost depends on the values of the arguments of the function. So, we need to carefully pick values for the arguments to cover most of the allowed ranges. Given that the allowed ranges are usually very wide (for double the range is $1.7E \pm 308$), smaller operating ranges for the built-in function being measured should either be specified beforehand (as a default), or entered by the user on-the-fly. Either the user accepts the default argument ranges or specifies other ranges. The points of the arguments should be selected uniformly in the ranges, i.e., equally spaced in the ranges to insure that we cover the whole range well in an unbiased manner. The measuring method is straightforward and is explained in the following figure—see the appendix for the experimental code.

Notes:

To avoid the storage of very large raw data set, we used a recursive method to calculate the mean and variance as follows: Assume that we have $N$ data points $\{x_i\}, i = 1, ..., N$. Also assume that we collect them index order. At step $i$ we can calculate the average of both the variable and its square:

$$Avg(i) = \frac{1}{i} \left( x_i + (i - 1) \cdot Avg(i - 1) \right)$$

$$AvgSQ(i) = \frac{1}{i} \left( x_i^2 + (i - 1) \cdot AvgSQ(i - 1) \right)$$

At the conclusion of the data collection, we can now specify:
\[
\text{Mean} = \text{Avg}(N) \\
\text{Variance} = \text{AvgSQ}(N) - \text{Avg}(N)^2
\]

```c
main ()
{
    GetRanges(); // ask user to input the range for each argument
    i = 1.0;      // Initialization
    PrevAvg = 0;
    PrevAvgSQ = 0;
    minTC = 100000000;
    maxTC = 0;
    for (Arg1=minArg1; Arg1<=maxArg1; Arg1 += (minArg1-maxArg1)/MAX_ITER)
    {
        for (Arg2=minArg2; Arg2<=maxArg2; Arg2 += (minArg2-maxArg2)/MAX_ITER)
        {
            ... 
            ...
            for (Argn=minArgn; Arg1<=maxArgn; Argn += (minArgn-maxArgn)/MAX_ITER)
            {
                TC = 100000000;
                for (j=0; j<MAX_REP; j++) // measure many repetitions and get
                {
                    tc = measureTC(ArgumentList); // from the environment
                    if (tc < minTC)
                        TC = tc;
                }
                Avg = (1/i) * ((double)TC + (i-1) * PrevAvg);
                PrevAvg = Avg;
                AvgSQ = (1/i) * ((double)TC*TC + (i-1) * PrevAvgSQ);
                PrevAvgSQ = AvgSQ;
                i++;
                if (TC < minTC)
                    minTC = TC;
                if (TC > maxTC)
                    maxTC = TC;
            }
            mean = Avg;
            var = AvgSQ - mean * mean;
        }
    }
}
```
In this approach for measuring the time cost of built-in functions, HPMS should have a table that contains (for each built-in function):

- The (default) ranges for the arguments (min point and max point).
- The mean (average) time cost of the built-in function.
- The variance of the time cost of the built-in function.
- The minimum of the time cost of the built-in function.
- The maximum of the time cost of the built-in function.

Moreover, HPMS should notify the user, whenever it encounters a built-in function, and display the above parameters for it—which were measured offline. Then, the user will have one of the following options:

1. Accept these defaults.
2. Specify other ranges for the arguments and re-measure the time cost on-the-fly.
3. Enter estimated values for the min, max and variance of the time cost of the built-in function directly.

3 Analytical approach

This section presents an approach for measuring the time cost of the built-in functions in the standard ANSI C language. First, we classify the built-in functions into the following categories:

- Zero-argument functions; e.g. getch(), …
- One-argument functions; e.g.: sqrt(double), atoi(char *), …
• Two-argument functions, e.g.: pow(double, double), strcpy(char*, char*), …
• Three-argument functions, e.g.: memcpy(void*, void*, size_t), …
• Four or more (fixed) argument functions; e.g: setvbuf (FILE*, char*, int, size_t), …
• Variable number of arguments functions; e.g.: printf(), scanf(),…

The basic idea of the approach is to approximate the time cost of a built-in function as a continuous polynomial function of its arguments. Then, we calculate the coefficients of this polynomial function and store them just as we did with the primitive operations. But here the time cost of the built-in function is measured using the calculated coefficients and the value(s) of the argument(s). We explain the approach for each case separately.

### 3.1 Zero-argument functions

These functions are very few. The time cost of these functions can be measured using the Performance Counters as we did in measuring the primitive operations. The time cost will be fixed (just as a primitive operation) since there is no arguments.

### 3.2 One-argument functions

Consider \(func(x)\) to be a generic name for all one-argument functions. The time cost of \(func(x)\) can be approximated by: \(t_{\text{func}(x)} = a_0 + a_1 x + a_2 x^2\). Additional terms can be included as necessary to improve the accuracy of the method.
To calculate $a_0$, $a_1$, and $a_2$:

- Pick three values for $x$: $x_{\text{min}}$ as the minimum value of $x$; $x_{\text{max}}$ as the maximum value of $x$; and $x_{\text{avg}} = (x_{\text{min}} + x_{\text{max}})/2$.

- Measure the time cost of $f(x_{\text{min}})$, $f(x_{\text{max}})$, and $f(x_{\text{avg}})$ and label them as $t_0$, $t_1$, and $t_2$, respectively. We measure these times using the Performance Counters method as we did in measuring the primitive operations—see the attached appendix for the experimental code.

- By substituting these values in the time cost equation we get three equations in three unknowns:

$$a_0 + a_1x_{\text{min}} + a_2x_{\text{min}}^2 = t_0$$
$$a_0 + a_1x_{\text{avg}} + a_2x_{\text{avg}}^2 = t_1$$
$$a_0 + a_1x_{\text{max}} + a_2x_{\text{max}}^2 = t_2$$

This can be represented in a matrix form as: $XA = T$, where:

$$X = \begin{bmatrix} 1 & x_{\text{min}} & x_{\text{min}}^2 \\ 1 & x_{\text{avg}} & x_{\text{avg}}^2 \\ 1 & x_{\text{max}} & x_{\text{max}}^2 \end{bmatrix}, \quad A = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}, \quad \text{and} \quad T = \begin{bmatrix} t_0 \\ t_1 \\ t_2 \end{bmatrix}$$

Hence, the coefficients are given by: $A = X^{-1}T$.

### 3.2.1 Example 1: sqrt(double x)

For this function, the time cost is always constant for any value of $x$ except when $x=0$.

We picked many random values in the range and the time cost is the same.

$x_{\text{min}} = 0$,

$x_{\text{avg}} = \text{any value (say 10)}$,

$x_{\text{max}} = \text{any value (say 100)}$. 
For the measurements to be as accurate as possible, we iterate the measuring process many times (say 100,000) and take the *minimum* time cost. Hence, we exclude almost all the interference from the running environment. Therefore, the *minimum* time cost for each point is:

\[ t_0 = 52 \text{ (cycles)} \]
\[ t_1 = 74 \text{ (cycles)} \]
\[ t_2 = 74 \text{ (cycles)} \]

Hence by using Matlab to solve, we get:

\[
A = \begin{bmatrix} 52 & 2.42 \\ -0.022 \end{bmatrix}.
\]

Note that, in this example the time cost is fixed with all points except at 0. This may suggest to treat this function and the similar ones as a special case and store only two values for the time cost (at 0 and at anywhere else.)

**3.2.2 Example 2: atoi(char *str)**

The time cost of this function is also a constant regardless of the argument length (including input of length 0). We tried many *randomly* chosen strings and the *minimum* time cost was fixed.

\[ x_{\text{min}} = \text{any value}, \]
\[ x_{\text{avg}} = \text{any value}, \]
\[ x_{\text{max}} = \text{any value}. \]

We created strings with lengths of \( x_{\text{min}}, x_{\text{avg}}, \text{and } x_{\text{max}} \). Then, we measured the time cost of each case:

\[ t_0 = 49 \text{ (cycles)} \]
$t_1 = 49$ (cycles)

$t_2 = 49$ (cycles)

Hence by using Matlab to solve: $A = X^{-1}T$, we get: $A = \begin{bmatrix} 49 \\ 0.0 \\ 0.0 \end{bmatrix}$.

### 3.3 Two-argument functions

Consider $\text{func}(x,y)$ to be a generic name for all two-argument functions. The time cost of $\text{func}(x,y)$ can be approximated by: $t_{\text{func}(x,y)} = a_0 + a_1 x + b_1 y + a_2 x^2 + b_2 y^2 + cxy$. In this case we have six unknowns: $a_0, a_1, a_2, b_1, b_2,$ and $c$. So, to calculate these unknowns do the following:

- Pick three values for $x$: $x_{\text{min}}$ as the minimum value of $x$, $x_{\text{max}}$, as the maximum value of $x$, and $x_{\text{avg}} = (x_{\text{min}} + x_{\text{max}})/2$.
- Pick three values for $y$: $y_{\text{min}}$ as the minimum value of $y$, $y_{\text{max}}$, as the maximum value of $y$, and $y_{\text{avg}} = (y_{\text{min}} + y_{\text{max}})/2$.
- Now, we have nine combinations of $x$ and $y$ as shown in the following table. We do recommend using the highlighted six to evaluate the unknowns since they better cover most of the possible cases than other combinations do.
Measure the time cost of \( f(x,y) \), for the highlighted six combinations and label them as \( t_0, t_1, t_2, \) as \( t_3, t_4, \) and \( t_5, \) respectively. We measure these times using the Performance Counters as we did in measuring the primitive operations.

By substituting these values in the time cost equation we get six equations in six unknowns:
This can be represented in a matrix form as: \( XA = T \), where:

\[
X = \begin{bmatrix}
1 & x_{\text{min}} & y_{\text{min}} & x_{\text{min}}^2 & y_{\text{min}}^2 & x_{\text{min}}y_{\text{min}} \\
1 & x_{\text{avg}} & y_{\text{avg}} & x_{\text{avg}}^2 & y_{\text{avg}}^2 & x_{\text{avg}}y_{\text{avg}} \\
1 & x_{\text{max}} & y_{\text{max}} & x_{\text{max}}^2 & y_{\text{max}}^2 & x_{\text{max}}y_{\text{max}} \\
1 & x_{\text{min}} & y_{\text{avg}} & x_{\text{min}}^2 & y_{\text{avg}}^2 & x_{\text{min}}y_{\text{avg}} \\
1 & x_{\text{avg}} & y_{\text{max}} & x_{\text{avg}}^2 & y_{\text{max}}^2 & x_{\text{avg}}y_{\text{max}} \\
1 & x_{\text{max}} & y_{\text{avg}} & x_{\text{max}}^2 & y_{\text{avg}}^2 & x_{\text{max}}y_{\text{avg}}
\end{bmatrix}, \quad A = \begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
b_1 \\
b_2 \\
c
\end{bmatrix}, \quad T = \begin{bmatrix}
t_0 \\
t_1 \\
t_2 \\
t_3 \\
t_4 \\
t_5
\end{bmatrix}
\]

Hence, the coefficients are given by: \( A = X^{-1}T \).

### 3.3.1 Example: pow(double, double)

Assume that the user specified the ranges for \( x \) and \( y \) as follows:

- Range for argument \( x \): min = 0 and max = 1000.
- Range for argument \( y \): min = 0 and max = 100.

Also, assume further that \( x_{\text{avg}} = (0+1000)/2 = 500 \) and \( y_{\text{avg}} = (0+100)/2 = 50 \). Therefore the \( X \) matrix will be:

\[
X = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 500 & 50 & 500^2 & 50^2 & 25000 \\
1 & 1000 & 100 & 1000^2 & 100^2 & 100000 \\
1 & 0 & 50 & 0 & 50^2 & 0 \\
1 & 500 & 100 & 500^2 & 100^2 & 50000 \\
1 & 1000 & 50 & 1000^2 & 50^2 & 50000
\end{bmatrix}
\]
By measuring the *minimum* time cost of each of these cases, we get the vector \( T = \begin{bmatrix} 31 \\ 287 \\ 287 \\ 107 \\ 287 \\ 287 \end{bmatrix} \).

Hence by using Matlab to solve \( A = X^{-1}T \), we get: \( A = \begin{bmatrix} 3.1000e + 001 \\ 5.4000e - 001 \\ 2.2800e + 000 \\ - 3.6000e - 004 \\ - 1.5200e - 002 \\ 0.0e - 001 \end{bmatrix} \).

### 3.3.2 Verification of the results

To assess to the accuracy of the technique we pick some random points (in the range of the arguments) then measure the real time cost and evaluate it from the approximated formula. Then, we calculate the error percentage between the measured and the evaluated time costs, as shown below.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Measured Time Cost</th>
<th>Estimated Time Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>287</td>
<td>287</td>
</tr>
<tr>
<td>5.5</td>
<td>0</td>
<td>228</td>
<td>33.959</td>
</tr>
<tr>
<td>0</td>
<td>30.4</td>
<td>89</td>
<td>86.265</td>
</tr>
<tr>
<td>890.44</td>
<td>34.9</td>
<td>315</td>
<td>287.46</td>
</tr>
<tr>
<td>76.09</td>
<td>59.78</td>
<td>316</td>
<td>151.98</td>
</tr>
<tr>
<td>530.78</td>
<td>97.45</td>
<td>314</td>
<td>294.04</td>
</tr>
</tbody>
</table>

We notices that there are some results that are not very accurate in the above table. Therefore, we need to refine the method. Many ideas were suggested to enhance the approach, such as partitioning of the ranges and modeling each range separately. Further research efforts are needed to get the desired results out of this approach.
3.4 Three-argument functions

.......................... TBD..........................

3.5 Variable number of arguments functions

.......................... TBD..........................

4 Appendix (experimental source code)

#include <stdio.h>
#include <conio.h>
#include <time.h>
#include <process.h>
#include <afx.h>
#include <sys/timeb.h>
#include <math.h>

#define CPUID __asm __emit 0fh __asm __emit 0a2h
#define RDTSC __asm __emit 0fh __asm __emit 031h
#define MAX_ITER 1000
#define MAX_REP 1000

/// This function returns the time cost of a built-in function,
// which is hardcoded in the module

unsigned long measureTC (double x, double y)
{
    unsigned long base=0;
    unsigned long cycles, cyc;
    float a=7, b=20, c=78;
    
    /// Set maximum Priority to this Thread
    int nOldPriority = GetThreadPriority(GetCurrentThread());
    BOOL ok = SetThreadPriority(GetCurrentThread(),
    THREAD_PRIORITY_TIME_CRITICAL);

    // The following tests run the basic cycle counter to find
    // the overhead associated with each cycle measurement.
    // It is run multiple times simply because the first call
    // to CPUID normally takes longer than subsequent calls.
    // Typically after the second run the results are
// consistent. It is run three times just to make sure.
__asm {
    ;cli //disable interrupt
    CPUID                 //first run
    RDTSC
    mov     cyc, eax
    CPUID
    RDTSC
    sub     eax, cyc
    mov     base, eax

    CPUID                 // second run
    RDTSC
    mov     cyc, eax
    CPUID
    RDTSC
    sub     eax, cyc
    mov     base, eax

    CPUID                 // third run
    RDTSC
    mov     cyc, eax
    CPUID
    RDTSC
    sub     eax, cyc
    mov     base, eax
    ;sti //enable interrupts
} // End inline assembly

/**** Measuring the cost of a built-in function
__asm {            // start measuring
    ;cli
    CPUID
    RDTSC
    mov     cycles, eax
}

// sqrt(x);       // function to be executed
// atoi(x);
// pow(x,y);
// log (x);
// cos(x);
// exp(x);
__asm {   // end measuring

    CPUID
    RDTSC
    sub    eax, cycles
    mov    cycles, eax
    ;     sti
}

/* printf("\n\n Cycle counts: %d, Measuring Overhead %d,\n\n\tCost of the function = %d\n\n",cycles, base, cycles-base);
   //printf("\n\nx=%e, \tfunc(x)=%e ", x, y);
*/

ok = SetThreadPriority(GetCurrentThread(), nOldPriority);
// retain the old priority

return ( cycles-base);    // return the net time cost
}   // end measureTC

void main (int argc, char** argv)
{

double x, y, minRangeX, maxRangeX, minRangeY, maxRangeY,
    PrevAvg, Avg, PrevAvgSQ, AvgSQ, mean, var, i;

unsigned long tc, j, TC, minTC, maxTC ;

    printf ("\n Input the Range (min  max) of the first argument: ");
    scanf("%lg %lg", &minRangeX, &maxRangeX);

    printf ("\n Input the Range (min  max) of the second argument: ");
    scanf("%lg %lg", &minRangeY, &maxRangeY);

    printf ("\n%g
%g
%g
%g
", minRangeX, maxRangeX, minRangeY,
    maxRangeY);

    // Pick uniformly spaced points for the arguments
    i = 1.0 ;
    PrevAvg = 0;
    PrevAvgSQ =0;
    minTC = 100000000;
    maxTC = 0;

    for (x=minRangeX; x<=maxRangeX; x+= (maxRangeX-minRangeX)/MAX_ITER)
    {
        for (y=minRangeY; y<=maxRangeY; y+= (maxRangeY-minRangeY)/MAX_ITER)
{  
    TC = 1000000000;
    for (j=0; j<MAX_REP; j++)   // Repeat and take minimum to exclude
        {                          //the interference from the environment
            tc = measureTC(x,y);
            if (tc < TC)
                TC = tc;
        }
    printf ("\n%ld", TC);
    Avg = (1/i) * ((double)TC + (i-1) * PrevAvg);
    PrevAvg = Avg;
    AvgSQ = (1/i) * ((double) TC*TC + (i-1) * PrevAvgSQ);
    PrevAvgSQ = AvgSQ;
    i++;
    if (TC < minTC)
        minTC = TC;
    if (TC > maxTC)
        maxTC = TC;
}

    } // end for
    } // end for

    mean = Avg;
    var = AvgSQ - mean * mean;

    printf ("\n Mean = %-12.2f Variance = %-12.2e Stand Dev =%-16.2f\n Minimum = %-12.2f\t Maximum= %-12.2f ", mean, var, sqrt(var),
            (double)minTC, (double)maxTC);
}

// end main