A FRAMEWORK FOR ACCURATE MEASUREMENTS WITH LOW RESOLUTION CLOCKS

José R. Herrero and Juan J. Navarro
Computer Architecture Department
Universitat Politècnica de Catalunya
Barcelona, Spain
email: {josepr,juanjo}@ac.upc.edu

ABSTRACT
The development of new efficient algorithms or implementations requires the validation of such codes as well as their timing. We need to measure accurately the execution time of a program so we can compare one algorithm with another. In this paper we present a framework we have devised which allows for accurate timings even with low resolution clocks. We have used this framework for the development of efficient numerical codes.

KEY WORDS
Benchmarking templates, low resolution clocks.

1 Introduction

The development of new efficient algorithms or implementations requires the validation of such codes as well as their timing. These tasks are often performed on different codes in a very similar way. In this paper we present a framework which we have devised and used for the development of efficient codes. We have been using this approach extensively in the development of numerical codes, in both dense [1] and sparse fields [2]. Using this approach we could automate the validation and benchmarking of a variety of codes, some of which were generated automatically [3].

This framework is based on a piece of code which is used as a template and can be configured with some preprocessor macros. The system provides a simple way to validate a new code. The user specifies an alternative way to produce the same operation: one which is known to be correct (a basic algorithm which can be taken as an oracle). Then, both codes are executed and results are validated: our system checks that the results obtained with both algorithms are equal.

Once we are sure that the new code is correct, we want to measure its performance. A problem can appear related to the precision of the timers. In this paper we present the way in which we handle this situation automatically. The process of measuring the performance becomes simple for the final user. He has to configure the system with a few preprocessor defines. Basically, he needs to specify the way to call the routine to be benchmarked, the one used for validating the results and the way the number of operations should be calculated.

1.1 Related work

It is well understood that collecting performance data on applications programs relying on timers with poor resolution or granularity is undesirable. Nowadays, most modern processors provide hardware counters which can provide accurate information about the performance of the applications. Thus, most performance tools rely on such counters. One problem with their use on different platforms comes from the different interfaces which have to be used. Fortunately, there have been several attempts to provide portable interfaces, like PCL [4] or PAPI [5]. Should any of them be available, we would advice using them. However, on some systems we may not have access to such tools. This can happen on rather outdated computers for which one still wants to automatically adapt some code, or on systems where such tools are not installed and one requires privileges, which do not have, for installation. For instance, to patch an operating system kernel. Under such circumstances we may be interested in using timers.

Standards have been created for timers and clocks [6]. However, regardless of the precision (milli, micro or nanoseconds), we can have a problem of lack of precision when the benchmarked code is executed very quickly and lasts for about the timer precision. Previous work was done in order to achieve high resolution timing with low resolution clocks [7]. Such high resolution can be achieved by repeated execution of a benchmark with a number of iterations through the code. Our approach resembles theirs. However, we provide a way to determine the number of iterations automatically.

Portable Timing Routines (PTR) [8] is a Ptools project defining a standard API for measuring intervals of program execution, in terms of wall-clock, user CPU, and system CPU time. We provide some routines similar to theirs which can ease the benchmarking process [9]. Our routines ensure accurate measurements with low resolution clocks.

2 Precision

An important issue when we have to time some code is the precision of the timing routine used. When the time needed to execute the routine is very small it can suffer from the lack of enough precision. In such cases we need

to execute such code repeatedly to get the total time. We need to determine the number of iterations of an algorithm necessary to obtain results to a desired precision.

2.1 Theoretical foundations

The speed of a numerical algorithm is usually communicated as the number of floating point operations performed per second. Often, this number is very large and is expressed in millions using the word *Mflops*.

$$Mflops = \frac{\#flops \cdot 10^{-6}}{Time}$$

The number of floating point operations (# flops) performed by an algorithm can be calculated by the programmer. The time spent in its execution can be obtained with some system calls offered by the operating system. Each of these system calls should have its precision clearly specified in the manual. Regardless of the precision provided by any of such routines it is finite. If the number of operations computed is very low, a lack of precision can occur.

Our goal is to get a correct estimation of the Mflops or execution time[1] obtained with a program. We need to be sure that the difference between the real and estimated Mflops is low. The variations in the estimated Mflops due to timing precision errors should be smaller than a certain threshold.

$$|\Delta Mflops| \leq Threshold \tag{1}$$

We can solve this problem by repeating the execution of a benchmark several times. By increasing the amount of operations we increase the time spent in their calculation. Consequently, the general expression for obtaining Mflops contains the number of iterations performed:

$$Mflops = \frac{\#flops \cdot Iterations \cdot 10^{-6}}{Time} \tag{2}$$

The number of iterations needed becomes an issue. We need to know the number of iterations which are necessary to obtain performance results which are correct. At the same time, we want to avoid unnecessary iterations which do not produce significant improvements in timing precision and would only overload the system and delay the finalization of our benchmarks.

We need an expression which can be used at execution time to reckon an adequate number of iterations for a given input program and data. To obtain it we have defined the following process. First, we get the derivative of equation 2 with respect to $Time$:

$$\frac{\partial Mflops}{\partial Time} = -\frac{\#flops \cdot Iterations \cdot 10^{-6}}{Time^2} \tag{3}$$

[1]We will center our discussion on the Mfbps metrics since that is the usual way to measure the speed of numerical algorithms.

We disregard the sign in equation 3 since we need only to get an small error (in absolute terms). To get a practical implementation we take the discrete analog of the derivative:

$$\frac{\Delta Mflops}{\Delta Time} = \frac{\#flops \cdot Iterations \cdot 10^{-6}}{Time^2} \tag{4}$$

From equation 2 we express $Time$ as a function dependent on the other components:

$$Time = \frac{\#flops \cdot Iterations}{Mflops \cdot 10^6}$$

Then, substituting $Time$ in equation 4 we obtain:

$$\frac{\Delta Mflops}{\Delta Time} = \frac{Mflops^2 \cdot 10^6}{\#flops \cdot Iterations}$$

Hence:

$$Iterations = \frac{Mflops^2 \cdot 10^6}{\#flops} \cdot \frac{\Delta Time}{\Delta Mflops} \tag{5}$$

Before a code is benchmarked we do not know its Mflops. In practice, we can use the peak theoretical value for the target machine. This will ensure that the results are correct. If we have an estimation of the Mflops of an algorithm we can provide that value in order to reduce the number of iterations.

2.2 Application

We can use equation 5 to determine the number of iterations necessary to obtain results within the desired precision. For instance, consider a timing routine with a precision of 10^{-2} seconds ($\Delta Time = 10^{-2}$). We want to obtain the number of iterations of a benchmarked subroutine such that the error in the estimation of its Mflops is approximately 1 Mflop ($\Delta Mflops \approx 1$):

$$Iterations = \frac{Mflops^2 \cdot 10^6}{\#flops} \cdot \frac{10^{-2}}{1} = \frac{Mflops^2 \cdot 10^4}{\#flops} \tag{6}$$

3 Framework

Figure 1 shows the files used and their relation. The system is mainly composed of a core file *profiler_template.c* which can be parameterized with some other files which are included via C *#include* directives. The files surrounded by dotted lines are supplied by the user. The rest of the files are provided by our system. Note that the benchmarked routine can be written in any language as long as it is callable from the C code we provide. The benchmarked routine can even be in a library provided at link time.

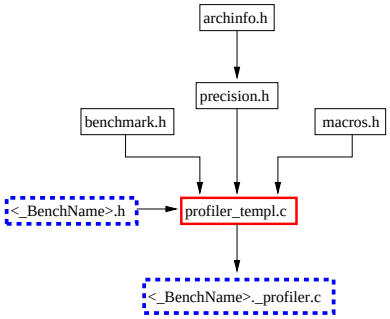

Figure 1. Template files and their relation

3.1 User files

When one wants to measure the performance of a new code and test it against another implementation, the user only has to write two additional files. A simple file which defines symbol *_BenchName* and includes the main template file *profiler_template.c*. This file is called *mtxms_profiler.c* in our example but could have any other name. Such file has the following form:

```
/* $Id: mtxms_profiler.c,v 1.1 2005/01/21 08:05:31
        myusername Exp $ */

#ifndef _BenchName
#define _BenchName mtxms
#endif

#include <profiler_templ.c>
```

The only other file the user must provide is *_BenchName.h* where _BenchName should be replaced by the benchmark name (*mtxms* in the example). This is the file used to customize the benchmarking. In it the user overrides the default macro definitions. An example follows:

```
/* $Id: mtxms.h,v 1.1 2005/01/21 08:05:31
        myusername Exp $ */

#ifndef _BenchRoutine
#define _BenchRoutine _BenchName
#endif
#ifndef _NUM_OPERATIONS
#define _NUM_OPERATIONS 2*i*j*k
#endif
#ifndef CALL_ROUTINE
#define CALL_ROUTINE \
      ad2(_BenchRoutine,_)( pdA, pdB, pdC )
#endif
#ifndef CALL_TEST_ROUTINE
#define CALL_TEST_ROUTINE \
      mtxms_test_ (pdA,pdB,pdD, &i,&j,&k, &lda,&ldb,&ldc)
#endif
#ifndef MATRIX_INITIALIZATION
#define MATRIX_INITIALIZATION \
      inimat_at_bn_(pdA,pdB,pdC,&i,&j,&k,&lda,&ldb,&ldc)
#endif

#ifndef CALL_GETINFO_BENCH
#define CALL_GETINFO_BENCH \
      ad3(getinfo_,_BenchRoutine,_)()
#endif
```

The user can use several variables declared in the system core file to denote matrices (pdA, pdB, pdC), their leading dimensions (lda, ldb, ldc) and the loop trip counts (i,j,k). Macros to compose names at compilation time (ad_2, ad_3, ...) are provided by our system. The above code defines the name of the routine to be benchmarked; the number of operations it performs; the way the routine has to be called; and the way the oracle routine has to be called. Only these definitions are compulsory.

Other symbols can be defined to allow for additional functionalities or modify the default behavior of the benchmarking system. For instance, we often include information about the parameters used at compilation time to create the executable. We can specify a routine which provides such information with CALL_GETINFO_BENCH. Defining symbol MATRIX_INITIALIZATION we can modify the default matrix initialization.

All the work necessary to drive the benchmarking is handled by the code supplied in the system files.

3.2 System files

Next, we present the most representative part of the files which constitute our framework.

3.2.1 profiler_templ.c

This file contains the template used as the main routine, the one which drives all the process. It is used to to launch benchmarks using several parameters. It is customized for a particular benchmark via a set of macros which can be defined in file _BenchName.h (where _BenchName is a preprocessor symbol which must be properly defined when cpp, the C preprocessor, is invoked).

Some symbols must be defined in _BenchName.h. In other cases the template file itself provides default values for these symbols, which can be overridden in the _BenchName.h file.

A pseudo-code with the most representative parts of this file follows:

```
/* $Id: profiler_templ.c,v 1.10 2004/11/04 15:51:27
        myusername Exp $ */

/* include header files */
...
#include <benchmark.h>
#include <macros.h>
#include <memory.h>
#include <precision.h>
...

#ifdef _BenchName
#define _FilNam <ad2(_BenchName,.h)>
#include _FilNam
#undef _FilNam
#endif

/* Default values for some preprocessor symbols */
#ifndef MATRIX_INITIALIZATION
#define MATRIX_INITIALIZATION \
      inimat_an_bt_(pdA,pdB,pdC,&i,&j,&k,&lda,&ldb,&ldc)
#endif
/* Some extra code is needed for testing some benchmarks
   but the default is not to need anything else. */
#ifndef EXTRA_DECLARATIONS
#define EXTRA_DECLARATIONS
#endif
```

```
#ifndef EXTRA_INITIALIZATIONS
#define EXTRA_INITIALIZATIONS
#endif
#ifndef EXTRA_FREEMEMORY
#define EXTRA_FREEMEMORY
#endif

#define EPSILON 10E-30

extern double validate_results_ ();

/* Global variables */
int     i, j, k, lda, ldb, ldc, it, ti;

main (argc, argv)
     int     argc;
     char   *argv[];
{
  declare_variables;
  get_parameters;
  initializations;

  it = GET_NUMITERATIONS (_NUM_OPERATIONS);

  Allocate_space;
  MATRIX_INITIALIZATION;

  if (check)
    {
      EXTRA_DECLARATIONS;

      EXTRA_INITIALIZATIONS;

      CALL_TEST_ROUTINE;

      CALL_ROUTINE;

      error = validate_results_ ( pdC, pdD, &i, &j, &ldc);
      if (error > EPSILON)
        {
          printf (
            xstr ( ERROR: _BenchRoutine test failed\n ) );
          printf ( "Error=%g\n", error );
          exit(-1);
        }
      else {
        printf (
          xstr ( OK: _BenchRoutine test succeeded\n ) ); }
      EXTRA_FREEMEMORY;
    }
  /* call BenchMarked routine  */

  GET_MFLOPS (CALL_ROUTINE, ti,it, _NUM_OPERATIONS, mflops);

  printf ("Mflops=%f Times=%d Iterations=%d \n",
          mflops, ti, it );

  Free_space;
  printf ("End of Execution\n");

}
```

Some of the definitions used in this file come from other files which are commented next. The details have been skipped for the sake of brevity.

3.2.2 macros.h

This file defines several macros for string manipulation which are useful in the creation of variable contents or routine names using preprocessor symbols at compilation time.

3.2.3 archinfo.h

_EX is used to specify the precision of the timing routine. PEAK_MFLOPS defines the theoretical peak performance which can be obtained on the target machine. This acts as an upper limit in the possible values for Mflops, which are not known a priori.

```
#define PEAK_MFLOPS 1000

/* Precision of timers:
 *
 * Considering precision as 10*1E-6
 *    #define _EX 1
 * Considering precision as 100*1E-6
 *    #define _EX 2
 */
#define _EX 2
```

3.2.4 precision.h

This file provides macros to reckon the number of iterations.

```
/* ---------------------------------------- DESCRIPTION
 * Define "Default Mflops" in order to reckon the
 * adequate number of iterations to obtain good accuracy
 */

/* ---------------------------------- FILES INCLUDED */
#include <math.h>
#include <archinfo.h>

/* ------------------------------------------- BEGIN */
#ifndef _DEFMFLOP
#define _DEFMFLOP PEAK_MFLOPS * .9
#endif

/*
 * Assuming an average of _DEFMFLOP Mflops per
 * algorithm, reckon # of iterations needed to
 * obtain enough precision: */

#define ITER_BASE \
   (_DEFMFLOP*_DEFMFLOP*(pow((double)10,(double)_EX)))

/* Example: for a MxM operation

    it= (int) ((unsigned long)
               (ITER_BASE/(unsigned long)(2*i*j*k)))+1;
    if (!it) it=1;
*/

#define GET_NUMITERATIONS(OPS) \
    (int) ((unsigned long)
           (ITER_BASE/(unsigned long)(OPS)))+1

#define DEBUG_PRECISION printf \
  ("DEFMFLOP=%f EX=%d IBASE=%f \n",
    _DEFMFLOP, _EX, ITER_BASE)

/* ---------------------------------------------- END */
```

3.2.5 benchmark.h

Here we define macros which handle the timing process. The user can easily get the best time or Mflops obtained by his routine amongst a number of repetitions.

```
 /* Defines macros:
 *   GET_BEST_TIME (what,times,iterations,best_time)
 *   GET_MFLOPS    (what,times,iterations,numops,mflops)
 */

/* Examples of usage:

 GET_BEST_TIME ( ad2(_MxMtName,_)( pdA, pdB, pdC ),
                 ti, it, best_time);
 GET_MFLOPS ( ad2(_MxMtName,_)( pdA, pdB, pdC ),
              ti, it, 2*i*j*k, mflops);
*/
```

4 Conclusions

We want to use hardware counters to do performance measurements whenever it is possible. However, in those cases where it is not, we still want to be able to obtain accurate performance measures using timers. To do so, we need to execute enough iterations to avoid problems of lack of precision. At the same time we do not want to perform unnecessary iterations. For a given desired precision of the result (Mflops in our case), we have shown how we can determine the number of iterations necessary to obtain it.

We have presented a framework for measuring the performance of new codes. It can be parameterized by the user to specify the way to call and test the routine being benchmarked. Then, our code will handle the benchmarking process. Depending on the problem size, it will automatically determine the adequate number of iterations.

Using this framework we have been able to validate and benchmark many linear algebra codes in a systematic way. We have been able to tune automatically our libraries for several platforms getting good performance on both dense and sparse codes.

Acknowledgement

This work was supported by the Ministerio de Educación y Ciencia of Spain (TIN2004-07739-C02-01).

References

[1] José R. Herrero and Juan J. Navarro. Compiler-optimized kernels: An efficient alternative to hand-coded inner kernels. In *Proceedings of the International Conference on Computational Science and its Applications (ICCSA). LNCS 3984*, pages 762–771, May 2006.

[2] José R. Herrero and Juan J. Navarro. Optimization of a statically partitioned hypermatrix sparse Cholesky factorization. In *Workshop on state-of-the-art in scientific computing (PARA'04),LNCS3732*, pages 798–807. Springer-Verlag, June 2004.

[3] José R. Herrero and Juan J. Navarro. Automatic benchmarking and optimization of codes: an experience with numerical kernels. In *Int. Conf. on Software Engineering Research and Practice*, pages 701–706. CSREA Press, June 2003.

[4] Rudolph Berrendorf and Heinz Ziegler. PCL the Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors.

[5] S Browne, J Dongarra, N Garner, G Ho, and P Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. of High Performance Computing Applications*, 14(3):189–204, 2000.

[6] ISO/IEC 9945-1:1996. [ANSI/IEEE Std 1003.1, 1996 Edition] Information technology - Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language].

[7] Peter B. Danzig and Stephen Melvin. High resolution timing with low resolution clocks and microsecond resolution timer for Sun workstations. *SIGOPS Oper. Syst. Rev.*, 24(1):23–26, 1990.

[8] PTR. Parallel tools consortium working group on portable timing routines.

[9] José R. Herrero and Juan J. Navarro. ACME: A framework for accurate measuments, 2006. http://personals.ac.upc.edu/josepr/Soft/ACME.