Performance Prediction through Time Measurements

Roman Iakymchuk

AICES, RWTH Aachen, Schinkelstr. 2, 52062 Aachen, Germany

iakymchuk@aices.rwth-aachen.de

Abstract. In this article we address the problem of predicting performance of linear algebra algorithms for small matrices. This approach is based on reducing the performance prediction to modeling the execution time of algorithms. The execution time of higher level algorithms like the LU factorization is predicted through modeling the computational time of the kernel linear algebra operations such as the BLAS subroutines. As the time measurements confirmed, the execution time of the BLAS subroutines has a piecewise-polynomial behavior. Therefore, the subroutines time is modeled by conducting only few samples and then applying polynomial interpolation. The validation of the approach is established by comparing the predicted execution time of the unblocked LU factorization, which is built on top of two BLAS subroutines, with the separately measured one. The applicability of the approach is illustrated through performance experiments on Intel and AMD processors.

Keywords

Linear algebra algorithms; BLAS subroutines; Performance prediction; Time measurements.

1 Introduction

Performance prediction is a challenging task that was tackled by dozens of scientists. Cuenca et al. [5] developed an analytical model to represent the execution time of self-optimizing parallel routines as a function of problem size, system and algorithmic parameters. In other works by Cuenca et al. [6, 7], the study was extended to the development of automatically tuned parallel linear algebra library; in those libraries the execution time of the higher-level routines like the LU factorization [9] is modeled using information generated from the lower-level routines such as the BLAS subroutines [10]. For instance, the execution time of a sequential blocked LU factorization for big problems ($n \geq 512$) is predicted using the optimal execution time of the BLAS subroutines. The optimal time is selected from a set of timings that are gathered during the installation of the automatically tuned library [8, 7]. Since timings are crucial for the performance prediction, Whaley et al. [12] investigated the importance of context-sensitive timers in achieving optimal performance of HPC applications. As a result, techniques and methodologies have been developed to obtain accurate timings of HPC kernel routines for the automated empirical tuning ATLAS framework [13].

In general, the performance of an algorithm can be defined as a ratio between the number of floating point operations performed by the algorithm and the execution time:

$$Performance = \frac{Number\ of\ FLOPS}{Execution\ time}. \tag{1}$$

For direct algorithms, *Number of FLOPS* can be calculated *a priori* from the structure of the algorithm. Since *Number of FLOPS* is known, the performance prediction reduces to modeling *Execution time*.

In this article, we analyze the impact of timings on the performance prediction and investigate modeling the execution time through measurements. Due to the accurate prediction for big problems ($n \geq 512$) [8] and the size of matrices in algorithms-by-blocks [4], our interest is in modeling the execution time of linear algebra algorithms for smaller problems ($n < 512$). We predict the execution time of higher level algorithms, like those included in the LAPACK library [1], through modeling the computational time of the BLAS subroutines. As the experiments confirmed, the execution time of the subroutines has a piecewise-polynomial behavior. Therefore,

we use a cycle-accurate timer to measure a subroutine for few problems and then model the subroutine time by applying polynomial interpolation. The approach is validated by predicting the execution time of an unblocked variant of an LU factorization [3] that is built on top of two BLAS subroutines, namely GER and SCAL. The performance experiments are conducted on the Intel and AMD processors.

The rest of the paper is structured as follows. Section 2 reviews timing methodologies and outlines the timing results. Section 3 describes the performance prediction. Section 4 presents the results of the evaluation. Section 5 provides conclusions.

2 Timing Methodologies

Modeling the execution time of linear algebra algorithms for small problems requires highly accurate timer in order to avoid the impact of inaccuracy in measurements on the output results. Usually, system timers that report time according to some system clock can be divided into two categories depending on whether they measure *CPU time* or *wall time*. Wall timers are very accurate with the resolution of clock cycle. The problem with wall timers is that they include the time spent on other users' tasks and unrelated OS operations. In general, wall timers provide a very accurate measure of elapsed time; although, a fraction of the kernel invocation in the timing results is unknown. Instead, CPU timers measure only time during which the processor is actively working on a particular task. Thus, CPU time does not include the time spent on other processes. However, the system aggregates time slices that are measured in terms of wall time to the CPU time of the appropriate process. Therefore, CPU time cannot have a greater resolution than the most accurate wall timer. CPU time is also highly inaccurate -- it can vary much from one measurement to another [12]. Because of this, in the experiments we prefer to use wall timer, especially our own cycle-accurate wall timer.

Timing does not only depend on the accuracy of the timer, but also on the placement of data that are used by an algorithm. This is specially the case for commercial and academic HPC libraries, like the BLAS library, where optimization and high-performance results (minimum execution time) are the main goals. To achieve these goals, libraries apply different timing approaches. However, the results of timing can be extremely misleading. For instance, while comparing the execution time of an algorithm from one library when operands are in the cache with the execution time of this algorithm from another library when operands are out of the cache. Therefore, the preferable approach for timers would be to flush the cache before timing an algorithm.

Another approach is when the kernel operations are timed in the way in which the algorithm calls them [12]. In our case an unblocked variant of an LU factorization, see Algorithm 1 [3], is built on top of two BLAS subroutines. Thus, to measure the execution time of Algorithm 1 we flush the cache only once before the first call of the BLAS subroutines; for the next calls operands would be partially in the cache. Thus, if there is no good reason to place operands in the cache, it is better to flush the cache before the computation.

To improve the accuracy of timing, we use multiple samples. For CPU time, the optimal timing in a set of modest samples is the *medium timing*. The medium timing is preferable than the average timing due to the oscillation in the measurements. Since the real source of inaccuracy of wall time is that other processes are included in the timings, the *most accurate timing* in terms of wall time is the *minimum timing* [12]. The results of applying different timing approaches are presented in the next section.

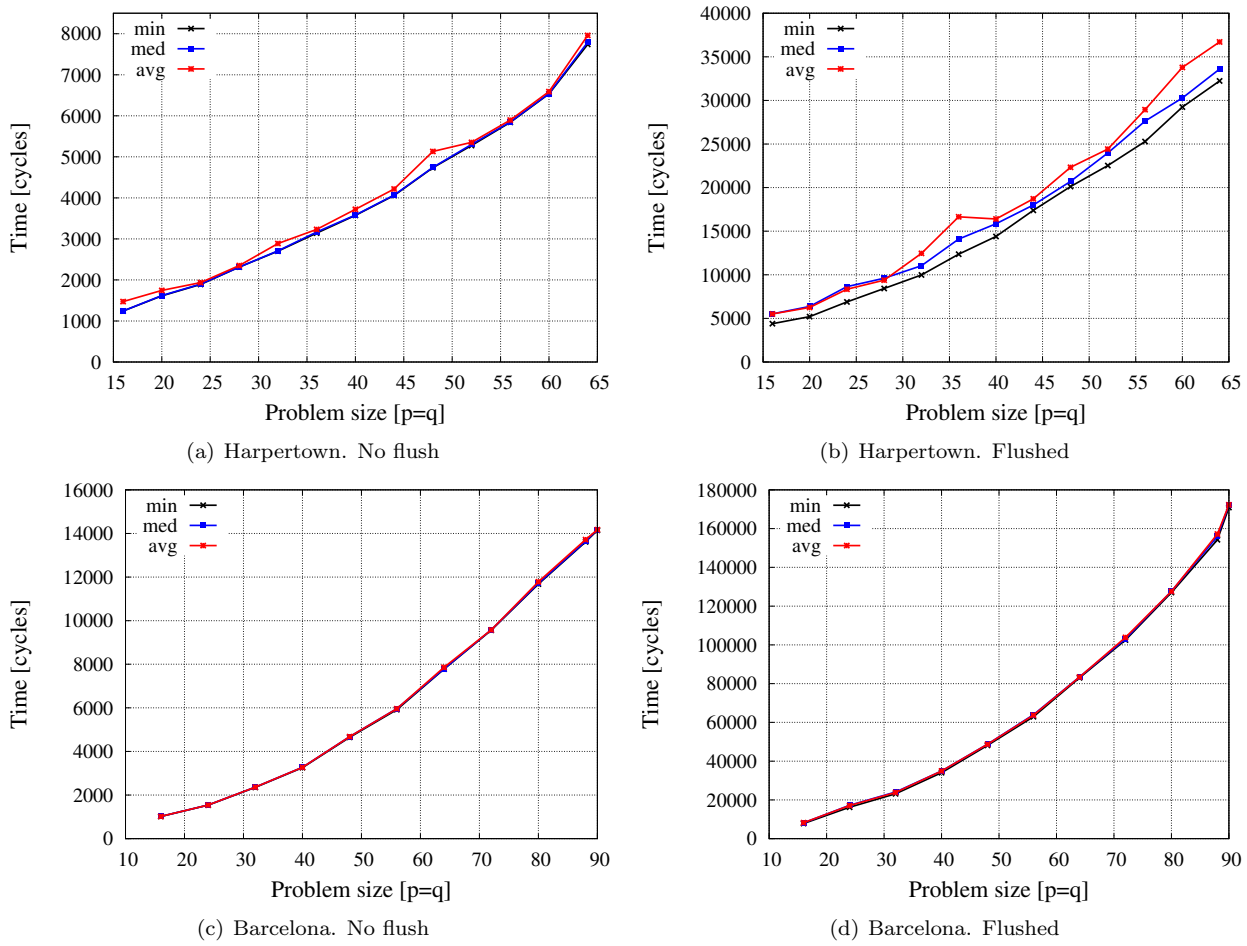
2.1 Time Measurements

The time measurements were performed on an Intel Xeon E5450 @ 2.4 GHz (code name Harpertown) and an AMD Opteron 8356 @ 2.3 GHz (code name Barcelona) processors. We report time measurements of GER [10], which is the BLAS level-2 subroutine that computes a rank-1 update of a general $m \times n$ matrix A:

$$A := \alpha x y^T + A, \tag{2}$$

where α is a scalar, x is a vector of size m and y is a vector of size n.

Fig. 1 shows the results of measuring GER on the Harpertown and Barcelona processors using two different timing methods. Fig. 1(a) and 1(c) present the results reported by naive timer which does not flush the cache before the computation. In contrast, Fig. 1(b) and 1(d) present the results reported by timer which ensures that operands are out of the cache. The flushed and no flushed timing results can differ significantly. For instance, on Harpertwon the flushed results are 4 times bigger than the no flushed ones. For each problem size, we conducted 10 samples of GER and then selected minimum, median and average timings from a set of samples. We discard the first value in the set since it includes time needed to cache instructions; it is mostly



(a) Harpertown. No flush

(b) Harpertown. Flushed

(c) Barcelona. No flush

(d) Barcelona. Flushed

Figure 1. Timing GER on the Harpertown and Barcelona processors.

greater than the other timings; and it can swamp the results. Figs. 1(a), 1(c), and 1(d) show that the distance between the minimum and medium timings are very small; however, as it is shown on Fig. 1(b) the median can be unstable and much bigger than the minimum. The average timing, which aggregates all the measurements except the first one, can vary considerably. Due to the usage of the cycle-accurate wall timer, we choose the minimum timing in the performance prediction experiments.

3 Performance Prediction

In this section, we apply the gained knowledge regarding timing approaches to the predict performance of linear algebra algorithms. The approach is demonstrated on Algorithm 1 [3], which represents an unblocked variant without pivoting for computing the LU factorization. Algorithm 1 decomposes a matrix A into a lower triangular matrix L and an upper triangular matrix U. The computation is performed by calling two BLAS subroutines [10]:

- SCAL -- a BLAS level-1 subroutine;
- GER -- a BLAS level-2 subroutine.

Algorithm 1 presents the unblocked variant of an LU factorization using FLAME notation [11, 2]. This notation does not operate with indexes and makes it easier to identify what regions of the matrix are updated and used, see Fig. 2. In Algorithm 1, $n(A)$ indicates the number of columns of the matrix A; 'T', 'B', 'L' and 'R' stand for 'Top', 'Bottom', 'Left' and 'Right', respectively. Before the computation starts, the matrix A is partitioned into four parts A_{TL}, A_{TR}, A_{BL} and A_{BR}, where A_{TL} is 0×0 matrix. The matrix A is traversed from the top left to the bottom right corner. At each iteration i of the loop the matrix A is repartitioned from 2×2 to 3×3 form. The algorithm updates the matrix A_{22} and the vector a_{21} using GER and SCAL subroutines,

Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

where A_{TL} is 0×0

While $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

where α_{11} is 1×1

$a_{21} := a_{21}/\alpha_{11}$ SCAL

$A_{22} := A_{22} - a_{21}a_{12}^T$ GER

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

endwhile

Algorithm 1. The LU factorization.

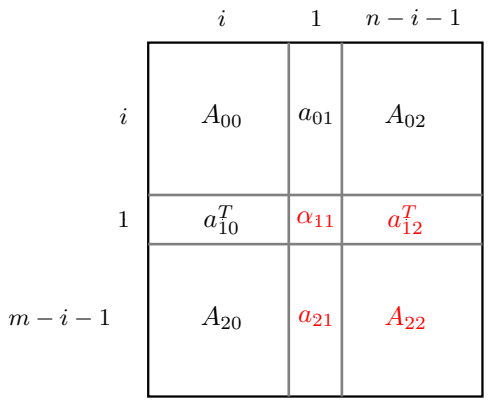


Figure 2. 3×3 partitioning of the matrix A.

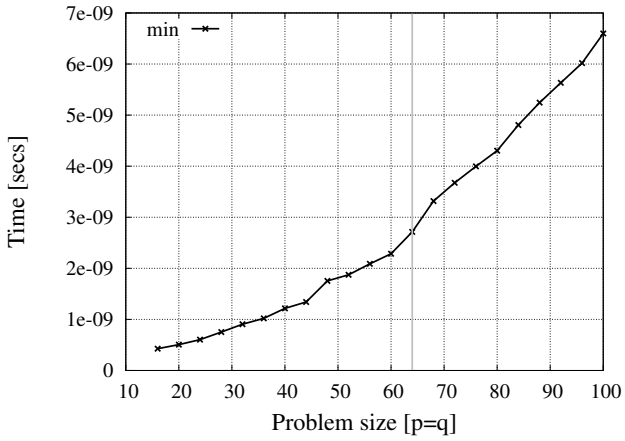
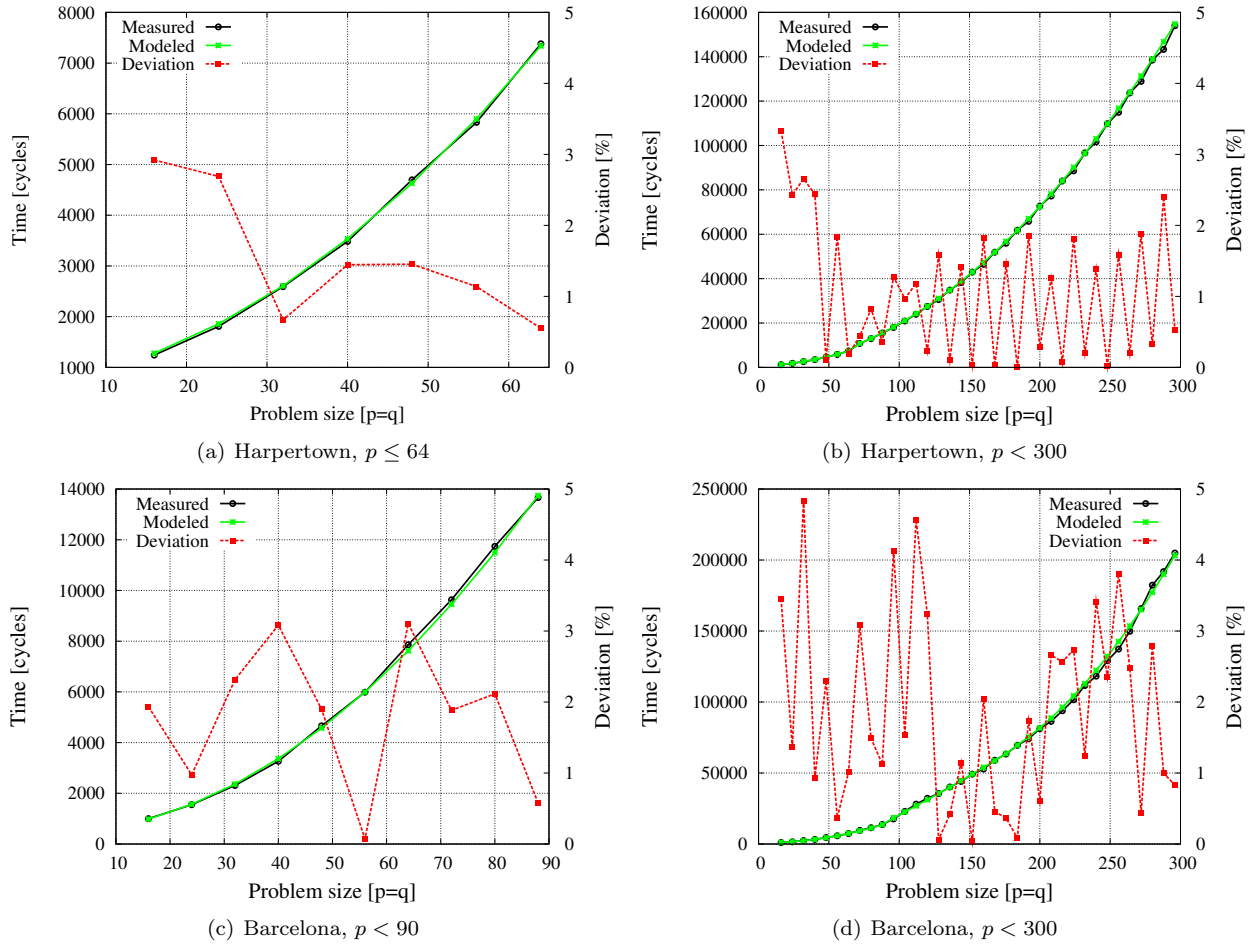


Figure 3. Piecewise-parabolic behavior of the GER execution time on Harpertown.

respectively. To denote the size of the matrix A_{22} we will use $p \times q$, where $p = m - i - 1$ and $q = m - i - 1$. As the algorithm progresses, the matrix A_{22} decreases in size starting from $m - 1 \times n - 1$ till 0×0. At the end of the computation the matrix A will be overwritten by the upper triangular matrix U and unit lower triangular matrix L.

Fig. 3 points out that the measured execution time of GER can be represented by a piecewise-parabolic function; the number of parabolas depends on the number of cache levels. For instance, the execution time for problems up to $p = 64$, which completely fit in the L1 cache on the Intel Harpertown processor, can be characterized by one parabola and for bigger problems, which fit in the L2, by another. Therefore, to model the GER time we evaluate GER only for few problems on each cache level, apply parabolic interpolation, and solve a linear least squares problem.

Since GER takes more than 96 % of the LU factorization execution time, we predict the LU factorization time by assembling the modeled time of GER.



(a) Harpertown, $p \leq 64$

(b) Harpertown, $p < 300$

(c) Barcelona, $p < 90$

(d) Barcelona, $p < 300$

Figure 4. Predicting the execution time of GER.

Table 1. Cache system on Intel Xeon E5450 (Harpertown).

Name	Total size	Line size	m ($m \times m$)
L1 data cache	32 KB	64 B	64
L2 unified cache	6144 KB	64 B	886

4 Evaluation

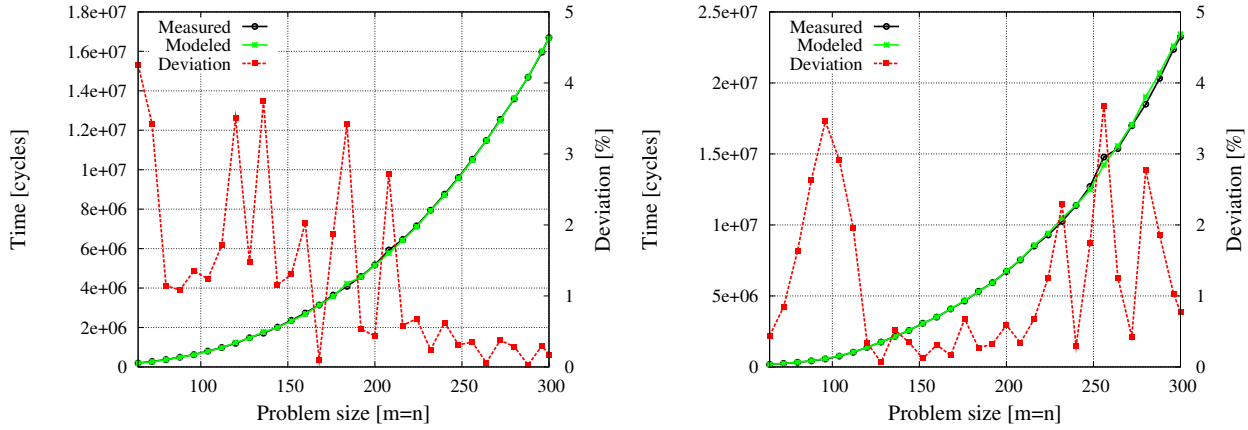
The performance experiments were performed on two different architectures: an Intel Harpertown and an AMD Barcelona processors. Each of the two Harpertown cores operates at 2.4 GHz and can execute 4 FLOPS per cycle, for a peak performance of 9.6 GFLOPS/sec per core. Memory-wise, each core includes a 32 KB L1 data cache and each chip has a shared 3 MB L2 unified cache. In addition, each chip has a 2 MB L3 unified cache shared among all four cores.

Barcelona, which is the second testbed for our experiments, has four cores. Each of them runs at 2.3 GHz and can execute 3 FLOPS per cycle for a peak performance of 6.9 GFLOPS/sec per core or 27.6 GFLOPS/sec per socket. Memory-wise, each core contains a 64 KB L1 data cache and a 512 KB L2 unified cache. In addition, each chip has a 2 MB L3 unified cache shared among all four cores. Tables 1-2 provide detailed information about cache systems on both processors. Since we are interested in modeling the execution time for small problems that mainly fit in the cache, Tables 1-2 also present matrix sizes that completely utilize the cache.

We verify modeling the execution time of GER and the LU factorization on the aforementioned architectures by comparing the predicted time with the separately measured one. For timings the cycle-accurate wall timer with flushing the cache was used. The time measurements were performed on GER from the highly optimized

Table 2. Cache system on AMD Opteron 8356 (Barcelona).

Name	Total size	Line size	m ($m \times m$)
L1 data cache	64 KB	64 B	90
L2 unified cache	512 KB	64 B	256
L3 unified cache	2048 KB	64 B	512



(a) Harpertown (b) Barcelona

Figure 5. Modeling the execution time of Algorithm 1.

GotoBLAS library [10]. The library was compiled with `-O2` flag; the `GOTO_NUM_THREADS` option was set to 1. We calculated the deviation (in percentage) between the modeled and the measured results by

$$Deviation = \frac{|Measured - Modeled|}{Measured} 100\%. \tag{3}$$

Fig. 4 presents the predicted and measured execution time of GER for different problem sizes on the Harpertown and Barcelona processors. To predict the GER time, we performed only $4 - 6$ measurements on each cache level, applied parabolic interpolation, and found coefficients of a parabola by solving linear least squares problems. Fig 4(a) and Fig. 4(c) show the results of modeling the execution time of GER when problems fit in the L1 cache on both processors. In that case, only one parabola is needed for the interpolation process; and therefore only one linear least squares problem is solved. In general, Figs. 4(a) -4(d) demonstrate that the time modeling is more accurate on Harpertown than on Barcelona. This is due to the three-levels cache system on Barcelona, so three parabolas are applied for modeling the execution time. On the whole, the deviation of the results is slightly high for small problems and problems that are close to the boundary between caches. However, mostly the deviation is less than 3 %.

The execution time of the unblocked variant of an LU factorization is modeled by assembling the predicted time of GER for $n - 1$ problems from $m - 1 \times n - 1$ down to 0×0. Figs. 5(a)-5(b) show the modeled and measured time of the unblocked variant of an LU factorization for different problem sizes on the Harpertown and Barcelona processors. Since for modeling the LU factorization time only GER is used, the deviation between the modeled and measured results is slightly higher for "tiny" problems. Nevertheless, the deviation is mainly less than 3 %; when the problem size increases the deviation converges to 0%.

5 Conclusions

We studied the general approach for predicting the performance of linear algebra algorithms through time measurements. The main focus of the study was on modeling the performance of algorithms that operate with small problem sizes. Due to the small problem sizes, the cycle-accurate wall timer was used in order to avoid the inaccuracy in the measurements. We started the study from the basic linear algebra algorithms like the BLAS subroutines. As the time measurements confirmed, the execution time of the BLAS subroutines has

a piecewise-polynomial behavior; a number of parabolas depends on the number of cache levels. Thus, the subroutines time is predicted by conducting only few measurements on each cache level and then applying polynomial interpolation. The execution time of higher level algorithms, which are built on top of the BLAS subroutines, is modeled by assembling the predicted time of the particular subroutines. The approach was validated by comparing the modeled and measured execution time of GER -- a BLAS level 2 subroutine -- and the unblocked variant of an LU factorization on the Intel and AMD processors. Finally, the study can be extended to modeling the execution time of the other BLAS subroutines and linear algebra algorithms that are built on top of them.

Acknowledgements

The authors gratefully acknowledge the support received from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111.

References

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In Proceedings of the 1990 ACM/IEEE conference on Supercomputing, pages 2–11. IEEE Press, 1990.

[2] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. ACM Transactions on Mathematical Software, 31(1):27–59, March 2005.

[3] Paolo Bientinesi and Robert A. van de Geijn. Representing dense linear algebra algorithms: A farewell to indices. FLAME Working Note #17. Technical Report TR-2006-10, The University of Texas at Austin, Department of Computer Sciences, February 2006.

[4] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, Salt Lake City, UT, USA, pages 123–132. ACM, 2008.

[5] Javier Cuenca, Domingo Giménez, and José González. Modeling the behaviour of linear algebra algorithms with message-passing. In Proceedings of the Euromicro Workshop on Parallel and Distributed Processing, Mantova, Italy, pages 282–289. IEEE Press, 2001.

[6] Javier Cuenca, Domingo Giménez, and José González. Towards the design of an automatically tuned linear algebra library. In Proceedings of the Euromicro Workshop on Parallel and Distributed Processing, Canary Islands, Spain, pages 201–208. IEEE Press, 2002.

[7] Javier Cuenca, Domingo Giménez, and José González. Architecture of an automatically tuned linear algebra library. Parallel Comput., 30(2):187–210, February 2004.

[8] Javier Cuenca, Domingo Giménez, José González, Jack Dongarra, and Kenneth Roche. Automatic optimisation of parallel linear algebra routines in systems with variable load. In Proceedings of the Euromicro Workshop on Parallel and Distributed Processing, Genova, Italy, pages 409–416. IEEE Press, 2003.

[9] Gene H. Golub and Charles F. Van Loan. Matrix computations (3rd ed.). Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[10] Kazushige Goto. GotoBLAS. Available via the WWW. `http://www.tacc.utexas.edu/resources/software/#blas`.

[11] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. ACM Transactions on Mathematical Software, 27(4):422–455, December 2001.

[12] R. Clint Whaley and Anthony M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. Softw., Pract. Exper., pages 1621–1642, 2008.

[13] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. Parallel Computing, 27(1-2):3–35, 2001.