Modeling Performance through Memory-Stalls

Roman Iakymchuk and Paolo Bientinesi
AICES, RWTH Aachen
Schinkelstr. 2
52062 Aachen, Germany
{iakymchuk,pauldj}@aices.rwth-aachen.de

ABSTRACT

We aim at modeling the performance of linear algebra algorithms without executing either the algorithms or any parts of them. The performance of an algorithm can be expressed in terms of the time spent on CPU execution and memory-stalls. The main concern of this paper is to build analytical models to accurately predict memory-stalls. The scenario in which data resides in the L2 cache is considered; with this assumption, only L1 cache misses occur. We construct an analytical formula for modeling the L1 cache misses of fundamental linear algebra operations such as those included in the Basic Linear Algebra Subprograms (BLAS) library. The number of cache misses occurring in higher-level algorithms—like a matrix factorization—is then predicted by combining the models for the appropriate BLAS subroutines. As case studies, we consider the LU factorization and GER—a BLAS operation and a building block for the LU factorization. We validate the models on both Intel and AMD processors, attaining remarkably accurate performance predictions.

Categories and Subject Descriptors

B.3.2.b [**Cache Memories**]; B.3.3 [**Performance Analysis and Design Aids**]; G.4.a [**Algorithm design and analysis**]

General Terms

Measurement, Performance

Keywords

Performance prediction, Performance model, Cache misses, Memory-stalls

1. INTRODUCTION

Predicting the performance of an algorithm is a problem that, although widely investigated, is still far from solved. Cuenca et al. [8] developed an analytical model to represent the execution time of self-optimizing individual routines as a function of problem size, system and algorithmic parameters. In another work by Cuenca et al. [9], the study was extended to the development of automatically tuned linear algebra libraries. The execution time of the higher-level routines is modeled by using information generated from the lower-level routines. Phansalkar et al. [14] proposed two simple techniques to predict performance and cache miss-rates based on the similarity of applications.

In contrast to the aforementioned works, our objective is to predict performance *without* executing either the target algorithm or parts of it. We first focus on the basic linear algebra operations like the ones included in the BLAS library [11]. For these, we develop analytical models that predict the amount of computation and data movement to be performed. The performance of higher-level algorithms, like those included in the LAPACK library [3], is then built by composing the models for the BLAS subroutines used within the algorithm.

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

$$Performance = \frac{FLOPS}{time(execution)}. \tag{1}$$

For direct algorithms, *FLOPS* can be calculated *a priori*; the prediction of the performance therefore reduces to the prediction of the execution time. In contrast to a model based on timing samples, we aim at obtaining the execution time by virtue of analytical formulas. This is of particular relevance in combination with techniques of automatic algorithm generation and tuning [4], in which dozens of algorithmic variants—written in terms of BLAS routines—have often to be evaluated. Once the models for BLAS are established, the prediction reduces to assembling and evaluating simple arithmetic formulas. In this paper we demonstrate the process by means of the LU factorization.

Our strategy consists of exploiting detailed information about the algorithm, the CPU, and the memory hierarchy. Since memory-stalls idle the CPU and add a significant overhead to the computational time, we model the computational time not only by means of the CPU execution time, but also through the time in which the CPU is idle due to memory-stalls. This last factor plays an especially important role in operations—like those included in the Levels 1 and 2 of BLAS, and the unblocked algorithms in the LAPACK library—that are memory-bound. Conversely, Level 3 BLAS operations are compute bound; for those, the execution time can be predicted rather accurately by a mere count of the floating point operations to be performed. For this reason, here we focus on the more challenging goal of predicting performance for memory-bound operations.

PMBS'11, November 13th, 2011, Seattle, Washington, USA.
ACM 978-1-4503-1102-1/11/11.

We restrict this study to a scenario where data resides in L2 cache, so only L1 data cache misses (L1 misses) occur. In our experiments, we use an unblocked variant of an LU factorization [5] that is built on top of the BLAS routines GER and SCAL [2]. SCAL only operates on one vector and its impact on the total number of floating point operations is negligible. Consequently, we model the cache misses for the unblocked LU factorization by modeling the misses for GER, for which we provide an analytic formula.

We validate our approach on two architectures with different processor types and memory systems. By working on two different architectures, we show that the basic formula needs to be tailored according to a number of parameters. Nevertheless, the advantage of the model is that it can be used for other algorithms with the same structure.

The paper is organized as follows. Section 2 reviews the main aspects of memory hierarchies. Section 3 describes the high-level performance model, while Section 4 introduces analytical models for L1 cache misses. We evaluate the models in Section 5, and discuss future work and conclusions in Sections 6 and 7, respectively.

2. MEMORY HIERARCHY

In a typical memory hierarchy, the storage devices get larger, slower and cheaper as we move from higher to lower levels. The highest level consists of several very fast CPU registers that can be accessed in one clock cycle. Next levels down are two or three small to moderate-size cache memories (cache levels or caches) that the CPU can access in few (between 3 and 20) clock cycles. The caches are followed by a large main memory that can be accessed in several dozens or few hundreds clock cycles [6].

The storage at each memory level k is partitioned into contiguous chunks called *blocks* (blocks in caches—*cache lines*). Copying data between memory levels is performed by blocks, e.g. between L2 and L1 caches by cache lines of 8 words. In order to know whether a piece of data is in cache and, if so, in which cache line, the hardware implements a mapping between the memory locations and the locations in the cache. The most widely used mapping is a *set associative* [6].

To proceed with the computation, a program needs to bring blocks of data from memory to registers. Before getting a block of data d from level $k+1$, the program first looks for d at level k. If d is cached at level k, then a *cache hit* occurs. The time needed to access the level k is called *cache hit time*, it includes the time needed to determine whether the access is a hit or not. On the other hand, if d is not found at level k, then a *cache miss* occurs. The time needed to serve the cache miss is called *cache miss time*, which is the time to fetch the block of data from level $k+1$ to level k and the delivery time of this block to the processor. Due to the smaller and the faster upper levels, the hit time is much smaller than the miss time [6, 13].

Our main focus is on modeling L1 misses for problems that reside in the L2 cache. Determining the number of L1 misses is a complex task that depends on the data access pattern of an algorithm as well as the associativity of the L1 cache and the processor type. For instance, a 2-way associative cache has more cache misses than 8-way associative cache [13].

One of the important features supported by processors is hardware prefetching. In general, prefetching means bringing data or instructions from the memory into the cache before they are needed. Usually hardware prefetching is very efficient, but in case of the data mis-prediction (when redundant data are prefetched) it pollutes the cache and causes unexpected cache misses [7]. The data access pattern of an algorithm is also crucial. The development of algorithms is mainly based on taking advantage of cached data. To exploit the temporal (data in cache) and spatial (data in cache line) localities, algorithms are adapted to the organization of the cache and the processor type. Thus, the organization of the cache, the processor type and the data access pattern of an algorithm are tightly coupled and altogether they dictate the behavior of L1 misses.

3. THE MODEL

A simple formula for modeling the execution time combines the number of clock cycles required by the algorithm and the duration of one clock cycle [13]:

$$\begin{aligned} time(execution) = {} & CPU\_clock\_cycles \\ & \times time(clock\_cycle). \end{aligned} \tag{2}$$

CPU_clock_cycles can be divided into the cycles that the CPU spends executing the algorithm and those caused by waiting for the memory system. Hence,

$$\begin{aligned} time(execution) = {} & [CPU\_execution\_clock\_cycles \\ & + Memory\text{-}stall\_clock\_cycles] \\ & \times time(clock\_cycle). \end{aligned} \tag{3}$$

The memory-stalls, caused by both cache and TLB misses, are divided into read-stalls and write-stalls. The write-stalls may occur without influencing the execution time, since data can be copied in the background. For this reason, in the following we only consider the read-stalls. Modeling the read-stalls can be a quite complex task that requires a deep understanding of the memory system behavior.

In order to obtain a more detailed model, the quantities *CPU_execution_clock_cycles* and *Memory-stall_clock_cycles* in Equation (3) should be refined into smaller components. *CPU_execution_clock_cycles*—which represents the actual computation (and includes the L1 cache hits)—can be written as follows

$$CPU\_execution\_clock\_cycles = FLOPS \times time(FLOP),$$

where *FLOPS* is the number of floating point operations performed. *Memory-stall_clock_cycles* can be split into 3 or 4 components depending on the number of cache levels in the memory hierarchy:

$$\begin{aligned} & Memory\text{-}stall\_clock\_cycles = \\ & \quad \sum_{i=1}^{n} L_i\_cache\_misses \times time(L_i\_cache\_miss) \\ & \quad + TLB\_misses \times time(TLB\_miss), \end{aligned}$$

where n is the number of cache levels. Thus, Equation (3) [13] becomes

$$\begin{aligned} & time(execution) = \\ & \quad \sum_{i=1}^{n} \alpha_i \times L_i\_misses \times time(L_i\_miss) \\ & \quad + \beta \times TLB\_misses \times time(TLB\_miss) \\ & \quad + \gamma \times FLOPS \times time(FLOP). \end{aligned} \tag{4}$$

The balancing weights α_i, β, γ are used to model the overlap between the computation and data movement. For compute-bound algorithms, $\gamma \approx 1$ and both α_i and β are close to zero. For memory-bound algorithms, the situation is reversed.

Our goal is analytical modeling of L1 misses. Being able to accurately model L1 misses when data resides in the L2 cache is the key to accurately model the execution time.

4. MODELING L1 MISSES

For our case studies we chose an unblocked variant of an LU factorization without pivoting, see Algorithm 1 [5]. Such algorithm is built on top of two BLAS subroutines [2], namely SCAL and GER. Our focus is on GER since it has quadratic complexity, as opposed to linear of SCAL, and is a major source of L1 misses. We will model the number of L1 misses for the LU factorization through the number of L1 misses for GER. We first consider GER in isolation, and then as part of the LU factorization.

```
Partition
  $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
      where $A_{TL}$ is $0 \times 0$
While $size(A_{TL}) < size(A)$ 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 $\alpha_{11}$ is $1 \times 1$

    $a_{21} := a_{21}/\alpha_{11}$            (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: An unblocked variant of the LU factorization.

GER is the BLAS level-2 subroutine that computes a rank-1 update of a general $m \times n$ matrix A:

$$A := \alpha x y^T + A, \tag{5}$$

where α is a scalar, x is a vector of size m and y is a vector of size n. GER performs $2mn$ floating point operations over $mn + m + n$ data, which means that $mn + m + n$ elements of the matrix and vectors have to be loaded into the CPU registers to proceed with the computation. Theoretically, each of those elements may cause an L1 miss. However, the elements of the matrix—stored by columns—and are *contiguous*, meaning that adjacent entries along one column are stored next to one another in memory. Since the data movement in the cache is organized by cache lines, only one cache miss per accessed cache line occurs.

To build a model for L1 misses, we make the following assumptions:

1. The matrix A and the vectors x and y^T are *aligned* to the beginning of a cache line—the first elements of the matrix and the vectors are the first elements in cache lines.
2. The L1 cache is large enough to keep both x and y^T and also some elements of the matrix A.
3. Once the vectors x and y^T are loaded into the L1 cache, they remain there for the whole computation.

With these assumptions, the number of L1 misses of GER can be modeled by

$$L1\_misses = \left\lceil \frac{mn}{d} \right\rceil + \left\lceil \frac{m}{d} \right\rceil + \left\lceil \frac{n}{d} \right\rceil, \tag{6}$$

where d is the number of double precision floating point values in a cache line; $\left\lceil \frac{mn}{d} \right\rceil$ is the number of L1 misses when reading the contiguous matrix A; $\left\lceil \frac{m}{d} \right\rceil$ and $\left\lceil \frac{n}{d} \right\rceil$ are the numbers of cache misses when reading the contiguous vectors x and y^T, respectively. Equation (6) covers only the simple scenario of the matrix and vectors being contiguous in the memory. In case of the unblocked variant of an LU factorization, the matrix A_{22} and vector a_{12}^T needed for GER are not contiguous anymore. Therefore, this influences the number of L1 misses and adds complexity to the model.

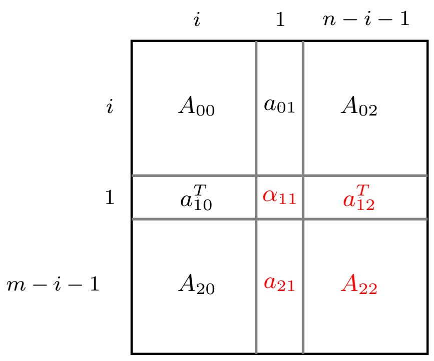


Figure 1: 3×3 partitioning of the matrix A.

The unblocked algorithm for computing the LU factorization is presented by Algorithm 1. The notation does not operate with indexes and makes it easier to identify what regions of the matrix are updated and used, see Figure 1. At each iteration i of the loop, the algorithm updates the matrix A_{22} and the vector a_{21} using SCAL and GER subroutines, respectively. To denote the size of the matrix A_{22} we will use $p \times q$, where $p = m - i - 1$ and $q = n - i - 1$. 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.

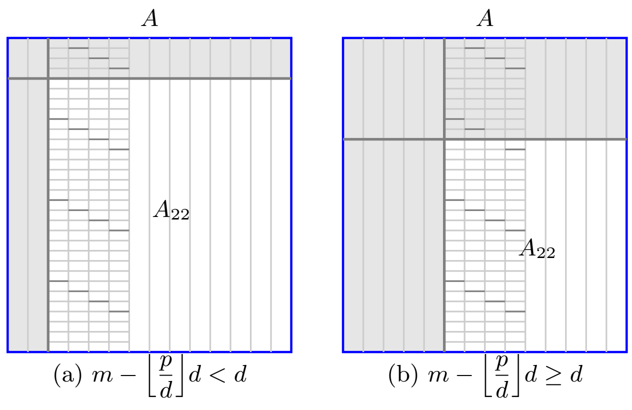
The storage of the matrices A and A_{22}, and the vectors a_{21} and a_{12}^T is organized as follows: A is stored by columns and it is contiguous in the memory; the leading dimension of A, which is denoted LDA, is the memory distance between the start of each row ($LDA = m$); A_{22}, which is the bottom-right part of A, is partially contiguous, see Figures 2(a) and 2(b); a_{21} is contiguous as it is the part of a column of A; and a_{12}^T is non-contiguous since each element of it belongs to a different column of A. As the matrix A_{22} is the major source of L1 misses in GER, we use a simplified case where the vector a_{12}^T is also contiguous. By taking this into account, Equation (6) can be modified as shown below

$$L1\_misses = \zeta + \left\lceil \frac{p}{d} \right\rceil + \left\lceil \frac{q}{d} \right\rceil. \tag{7}$$

The quantity

$$\zeta = \begin{cases} \left\lfloor \frac{mq}{d} \right\rfloor, & \text{if } m - \left\lfloor \frac{p}{d} \right\rfloor d < d \\ \left\lceil \frac{p}{d} \right\rceil + \sum_{i=1}^{n-1} \left\lceil \frac{p + (mi \bmod d)}{d} \right\rceil, & \text{otherwise} \end{cases} \tag{8}$$

indicates L1 misses when loading the matrix A_{22} from the L2 cache. The second case in Equation (8) represents cache misses when reading each column of the matrix A_{22} including a "tail" $(mi \bmod d)$ from the previous column, see Figure 2(b).



A
A_{22}
(a) $m - \left\lfloor \frac{p}{d} \right\rfloor d < d$
A
A_{22}
(b) $m - \left\lfloor \frac{p}{d} \right\rfloor d \geq d$

Figure 2: Alignment of cache lines within the matrix A_{22}.

Equation (8) with two cases covers all two possible scenarios of the matrix A_{22} being a part of the matrix A. These two scenarios are illustrated by two plots in Figure 2. On each of these plots, the storage of the matrix A and partitioning of the elements of A into cache lines (8 elements per a cache line) is shown. The first figure, Figure 2(a), represents the situation when the matrix A_{22} is the bottom-right part of A and $m - \left\lfloor \frac{p}{d} \right\rfloor d < d$. Thus, the whole matrix A_{22} and redundant data between two columns of A_{22} will be loaded. Accordingly, we use $\left\lfloor \frac{mq}{d} \right\rfloor$ to model L1 misses on reading A_{22}. The second figure, Figure 2(b), illustrates the situation when $m - \left\lfloor \frac{p}{d} \right\rfloor d \geq d$. Therefore, only some redundant data between two columns of the matrix A_{22} will be loaded as it is represented by $(mi \bmod d)$ in the second case of Equation (8).

In the next two sections we show how the basic model for predicting L1 misses, Equations (7) and (8), captures the behavior of GER on two different architectures: AMD Barcelona and Intel Penryn.

4.1 AMD Barcelona

For our experiments we used an AMD Opteron 8356 processor (code name Barcelona). Each of the four cores runs at 2.3 GHz and can execute 3 FLOPS per cycle for a peak performance of 6.9 GFLOPS/s per core or 27.6 GFLOPS/s per socket. Memory-wise, each core contains a 64 KB L1 data cache and a 512 KB L2 unified cache[1]. In addition, each chip has a 2 MB L3 unified cache shared among all four cores.

[1]A cache is said to be "unified" when data and instructions can be stored.

Barcelona has two prefetching units: one for the code and another for the data. The data prefetching unit fetches data only into the L2 cache [12]. Thus, the mis-prediction of hardware prefetching does not play a role in modeling L1 misses. Accordingly, the modeling on Barcelona can be done by Equations (7) and (8).

4.2 Intel Penryn

The Intel Core 2 P8600 (code name Penryn) is the second testbed for our experiments. Each of the two cores operates at 2.4 GHz, and can execute 4 FLOPS per cycle, for a peak performance of 9.6 GFLOPS/s per core. Memory-wise, each core includes a 32 KB L1 data cache and each chip has a shared 3 MB L2 unified cache.

Our interest in the Intel Penryn processor is due to the different organization of the cache system with respect to the AMD Barcelona processor. Also, unlike Barcelona, Penryn has a powerful prefetching mechanism: a dedicated unit detects multiple reads from a single cache line and loads the next line into the L1 cache [10]. Each access to the L2 cache due to the prefetching counts as an L1 misses by processor's performance counters. Prefetching may also lead to the mis-prediction of data that causes unexpected L1 misses. By including L1 misses due to the mis-prediction of hardware prefetching into the model, Equations (7) and (8) can be rewritten as

$$\mathit{L1\_misses} = \zeta' + \left\lceil \frac{p}{d} \right\rceil + \left\lceil \frac{q}{d} \right\rceil + 3, \tag{9}$$

where

$$\zeta' = \begin{cases} \left\lfloor \frac{mq}{d} \right\rfloor, & \text{if } m - \left\lfloor \frac{p}{d} \right\rfloor d < d \\ \left\lceil \frac{p}{d} \right\rceil + \sum_{i=1}^{n-1} \left(\left\lceil \frac{p + (mi \bmod d)}{d} \right\rceil + \eta(i) \right), & \text{otherwise.} \end{cases} \tag{10}$$

In Equation (9), due to the prefetching of 1 extra cache line after the load of an operand, three misses are added: 1 for the matrix and 2 for the vectors. In the second case of Equation (10), cache misses caused by the mis-prediction in hardware prefetching are also included. For instance, $\eta(i)$, which represents the number of L1 misses per each column of the matrix A_{22} due to the mis-prediction, is given as

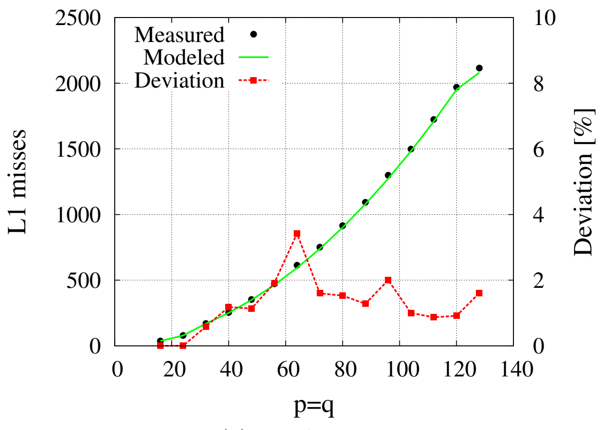
$$\eta(i) = \begin{cases} 1, & \text{if } \left\lfloor \frac{m + (mi \bmod d)}{d} \right\rfloor - \left\lceil \frac{p + (mi \bmod d)}{d} \right\rceil > 0 \\ 0, & \text{otherwise.} \end{cases}$$

5. EVALUATION

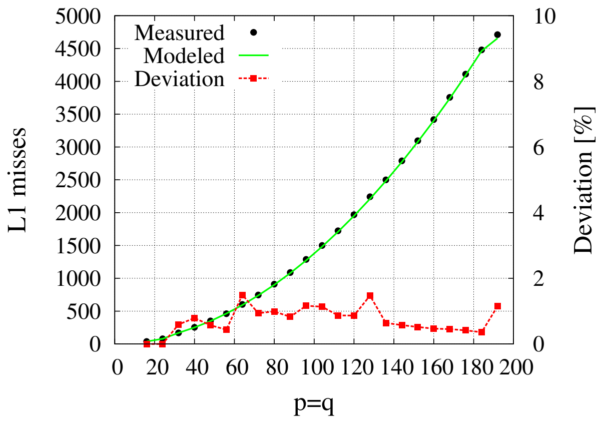
In this section we verify the accuracy of the models for GER and the LU factorization by comparing with the number of misses measured in practice.

The experiments were performed on two implementations of GER: one from the reference BLAS library [2] and the other from the highly optimized GotoBLAS2 library of version 1.07 [11]. The libraries were compiled with the `-O2` optimization flag. The `GOTO_NUM_ THREADS` variable was set to one, i.e., only one core was used in all experiments. The PAPI library [1] was used to access the processor's performance counters. We calculated the deviation (percentage) between the modeled and the measured results by

$$\mathit{Deviation} = \frac{|\mathit{Measured} - \mathit{Modeled}|}{\mathit{Measured}} 100\%. \tag{11}$$



(a) $LDA = 128$



(b) $LDA = 192$

Figure 3: Deviation between predicted and measured L1 misses for GER on Barcelona.

5.1 AMD Barcelona

Figure 3 reports the accuracy of our model for L1 misses on the AMD Barcelona processor. For GER from the reference BLAS, we plot the expected and measured misses, as well as the relative deviation, for problems of size $p = q$ and leading dimension m. We chose the problem size in the interval $[16, 256]$ as a typical size for the unblocked variant (Algorithm 1) of an LU factorization. The unblocked variant in fact operates with problems that are small enough to fit in the L2 cache memory. Moreover, we concentrate on small sizes because—as our results suggest—the model for L1 misses becomes more accurate as the problem size increases. The results of the model are always satisfactorily accurate, even for small problems; in most cases, the deviation is below 2%.

In order to simulate the behavior of GER within Algorithm 1, we fix m and vary the problem size ($p = q$) from $p = m$ and down to $p = 16$. Figures 3(a) and 3(b) show the results for $LDA = 128$ and 192, respectively.

Figure 4 refers to the more general scenario of $p \geq q$, which covers the shapes of A most commonly encountered in a factorization. In the figure, $LDA = 512$. As for the case $p = q$, the deviation is small, usually less then 2%. The only exception is for problems of very small size (the

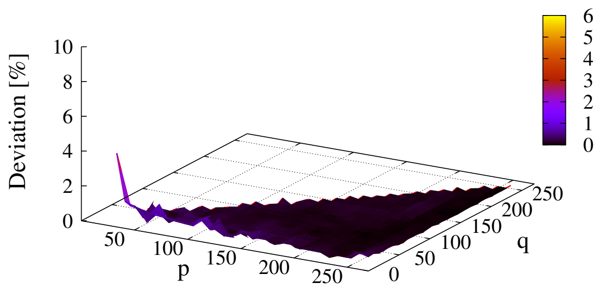


Figure 4: Deviation between predicted and measured L1 misses for GER on Barcelona; $p \geq q$, $LDA = 512$.

Table 1: Deviation between modeled and measured L1 misses for the unblocked LU factorization on Barcelona.

m=LDA	Modeled	Measured	Deviation[%]
64	12,601	12,612	0.087
96	40,977	41,306	0.797
128	94,889	95,899	1.053
192	312,089	313,926	0.585
256	729,737	732,501	0.377

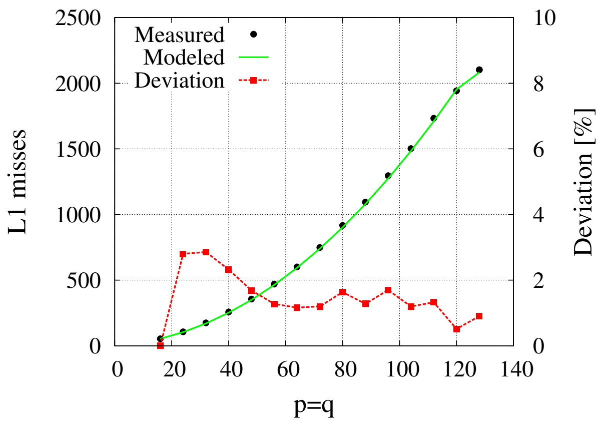
deviation is around 4%), which anyway carry little or no weight at all when considered as part of a factorization. This statement is confirmed by the numbers in Table 1: Since our focus is on GER as the computational kernel for the LU factorization, we are interested in the sequence of problems ($p = q$) from the interval $[1, m]$. In Table 1, we add the number of modeled L1 misses of GER for different values m. The result compared with the number of misses separately measured during an LU factorization. In this scenario, the model attains an even higher level of accuracy: the deviation is mostly less than 1%.

5.2 Intel Penryn

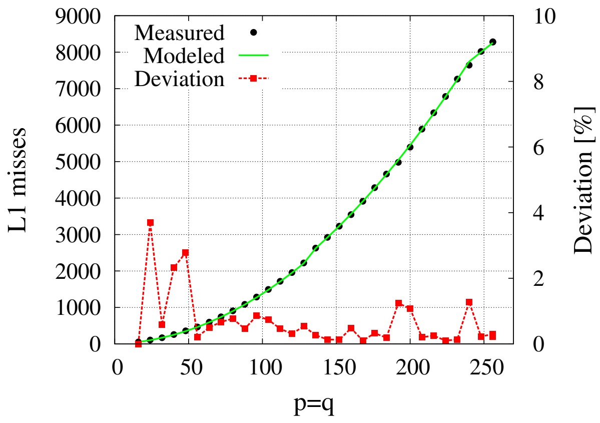
Figures 5(a) and 5(b) present the expected and measured L1 misses for GER from the reference BLAS library on the Intel Penryn processor. The model shows satisfactorily results; the deviation is mostly less than $2 - 3\%$. In the rest of the section our focus is only on results for the GotoBLAS2 library. Its level of sophistication, together with the fact that we did not attempt to study its source code, makes this a more difficult object to model.

The case $p = q$ is illustrated in Figures 6(a) and 6(b) for $LDA = 128$ and 256, respectively. The deviation is larger for problems of size smaller than $p = 40$. This is due to the small number of L1 data cache misses; even a slight difference of 2-6 cache misses between the model and the measurement becomes significant in terms of deviation. In general, for most problems the deviation is less than 3%. In summary, except for the smallest problem sizes, the model is very accurate.

We observe spike in the deviation, especially, for small

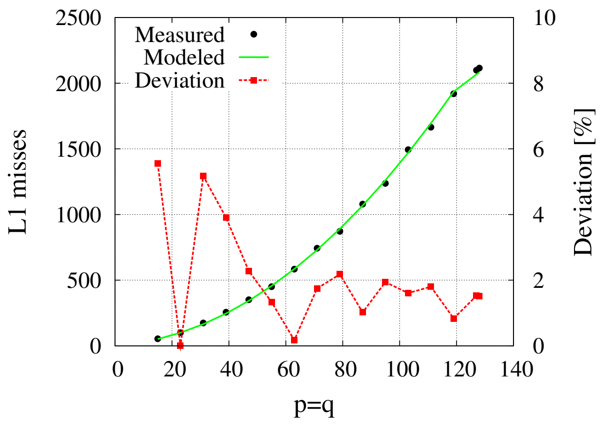


(a) $LDA = 128$

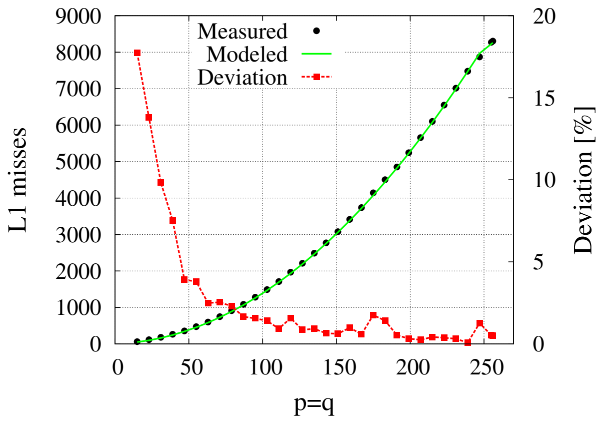


(b) $LDA = 256$

Figure 5: Deviation between predicted and measured L1 misses for GER from the reference BLAS library on Penryn.



(a) $LDA = 128$



(b) $LDA = 256$

Figure 6: Deviation between predicted and measured L1 misses for GER from the GotoBLAS2 library on Penryn.

problems. However, for the most cases the deviation is less than 3%. From the all results we can summarize that the modeling L1 misses is very accurate for relatively big problems. These problems have a much higher number of cache misses than small ones, so they are a major source of L1 misses in Algorithm 1.

In Table 2, we model the number of misses for the LU factorization by adding up all the predictions of GER. The deviation of the combined results is mostly less than 2%. This demonstrates that the relatively high deviation for small problem sizes is negligible in the context of a factorization.

Figure 7 presents the results of modeling L1 misses of GER in the more general scenario $p \geq q$ ($LDA = 512$). The figure indicates that in the area close to the origin the deviation is slightly higher than for the rest of the spectrum. However, as demonstrated above, such deviations do not affect the accuracy of the model for the whole LU factorization.

Finally, we look at the scenario in which the unblocked algorithm, Algorithm 1, is a building block for a blocked variant of an LU factorization; in this case, q is small, $p \geq q$, and LDA might be much greater than p. To model the L1 misses of Algorithm 1, we aggregate the models for GER from of size $p \times q$, $p-1 \times q-1$, down to $p-q+1 \times 1$. Figure 8 shows the modeled and measured L1 misses for $LDA = 2048$ and two fixed q ($q = 64$ and 128). Since q is fixed and only p varies, the modeled and measured misses demonstrate linear behavior. In summary, the results are very accurate—the model is within $2 - 4\%$ of the measurements.

6. FUTURE WORK

The work described in this paper represents a successful first step towards the ultimate goal of predicting the performance of linear algebra computations. In order to gain applicability, the model has to be extended in a number of ways. First off, larger problems will not fit in the L2 cache, therefore causing not only L1 but also L2 misses. Lifting memory constraints and incorporating L2 cache misses into the model is the natural next step; for even larger problems, TLB misses must be taken into account too. In this document we only present models for one BLAS operation; many others building blocks are needed. Conceptually our model can be adapted to other level 1 and level 2 BLAS operations without major hurdles. The modeling for level 3 operations is substantially different, as those operations are CPU-bound; as a consequence, data movement and cache misses are of secondary importance for accurate timing predictions.

The reader might be wondering whether the accurate prediction of cache misses translates to accurate timing predic-

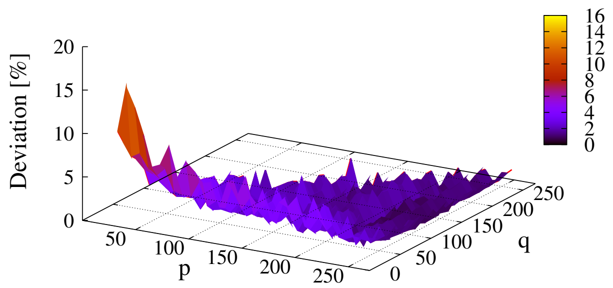


Figure 7: Deviation between predicted and measured L1 misses for GER from the GotoBLAS2 library on Penryn; $p \geq q$, $LDA = 512$.

Table 2: Deviation between modeled and measured L1 misses for the unblocked LU factorization on Penryn.

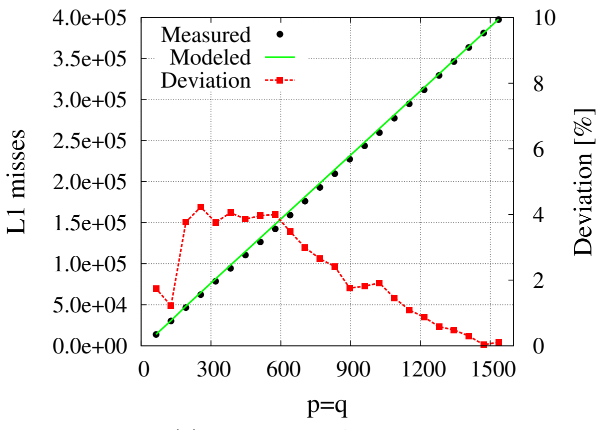
m=LDA	Modeled	Measured	Deviation[%]
64	14,009	13,500	3.770
96	44,433	43,698	1.682
128	101,289	100,643	0.642
192	327,065	326,145	0.282
256	756,822	753,946	0.382

tion. Our preliminary results suggest a positive answer: In Figure 9 we show predicted and measured execution time of GER when the operands reside in the L2 cache. For the prediction, we used Equation (4) in conjunction with the model for L1 misses and a suitable choice of the parameters α_1, and γ.[2] The outcome is quite encouraging. The deviation is always below 6%, even for small problems, and in most cases is around or lower than 2%.

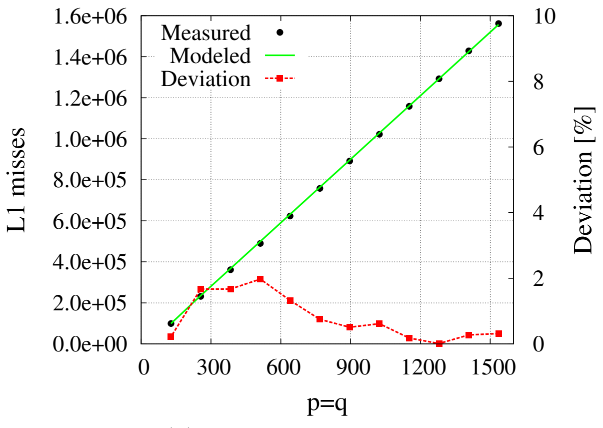
7. CONCLUSIONS

We set out to predict the performance of linear algebra algorithms without any actual code execution. In fact, we proposed an analytical model solely based on detailed knowledge of the algorithm as well as the CPU and the memory hierarchy. As it was shown, modeling performance is equivalent to modeling both the CPU execution time and the time spent on memory-stalls. This paper focuses mainly on memory-stalls. We considered the scenario in which the input data resides in the L2 cache, and we built an analytical formula for modeling L1 cache misses. As target algorithms, we considered kernel linear algebra operations like those included in the BLAS library. To verify the model, we conducted a set of experiments using GER from the reference BLAS and the GotoBLAS2 libraries. By working on two architectures, we tailored the basic formula for different processor types and memory systems. In all cases, the model resulted extremely accurate, with deviations normally lower than 2%. We chose the GER kernel because it is responsible for most of the computation of an unblocked variant of an LU factorization. Thus, by composing modeled L1 misses of GER, we predicted the number of misses for the LU factorization. A comparison between the model and the actual measurements yielded deviations below 3%. Finally, we presented initial evidence that an accurate model for cache misses leads to accurate prediction for the execution time.

[2]We set both α_2 and β to zero as the problems reside in the L2 cache.



(a) $q = 64$, $LDA = 2048$



(b) $q = 128$, $LDA = 2048$

Figure 8: Deviation between predicted and measured L1 misses for Algorithm 1 when used as part of a blocked LU factorization.

Acknowledgments

We would like to thank Prof. Enrique S. Quintana Ortí and Diego Fabregat Traver for their advices and many useful discussions.

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

8. REFERENCES

[1] Performance Application Programming Interface (PAPI). Available via the WWW. `http://icl.cs.utk.edu/papi/`.

[2] The reference BLAS. Available via the WWW. `http://www.netlib.org/blas/`.

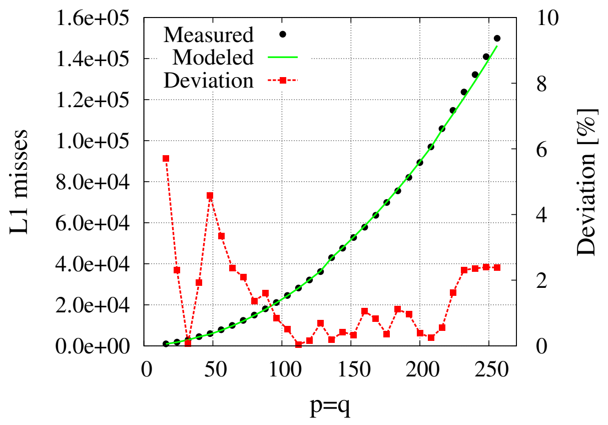


Figure 9: Deviation between predicted and measured execution time of GER from the GotoBLAS2 library on Penryn; $LDA = 512$.

[3] 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*, Supercomputing '90, pages 2–11. IEEE Computer Society Press, 1990.

[4] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. Families of algorithms related to the inversion of a Symmetric Positive Definite matrix. *ACM Trans. Math. Softw.*, 35(1):1–22, 2008.

[5] 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.

[6] Randal E. Bryant and David R. O'Hallaron. *Computer Systems – a Programmers Perspective*. Pearson Education, 2011.

[7] Shannon Cepeda. What you need to know about prefetching, August 2009. Available via the WWW. `http://software.intel.com/en-us/blogs/2009/08/24/`.

[8] 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.

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

[10] Franck Delattre and Marc Prieur. Intel Core 2 Duo – test, July 2006. Available via the WWW. `http://www.behardware.com/articles/623-6/`.

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

[12] Yury Malich. AMD K10 micro-architecture, August 2007. Available via the WWW. `http://www.xbitlabs.com/articles/cpu/display/amd-k10_9.html`.

[13] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface, Fourth Edition*. Morgan Kaufmann, 2009.

[14] Aashish Phansalkar and Lizy K. John. Performance prediction using program similarity. In *Proceedings of the 2006 SPEC Benchmark Workshop*, January 2006.