mathematics

Article

Ideal and Predictable Hit Ratio for Matrix Transposition in Data Caches

Alba Pedro-Zapater [1,*], Clemente Rodríguez [2], Juan Segarra [1], Rubén Gran Tejero [1] and Víctor Viñals-Yúfera [1]

[1] Departamento de Informática e Ing. de Sist., I3A, Universidad de Zaragoza, 50009 Zaragoza, Spain; HiPEAC; jsegarra@unizar.es (J.S.); rgran@unizar.es (R.G.T.); victor@unizar.es (V.V.-Y.)
[2] Departamento de Arquitectura y Tecnología de Computadores, Universidad del País Vasco, 48940 Leioa, Spain; acprolac@ehu.es
* Correspondence: albapz@unizar.es

Received: 27 December 2019; Accepted: 23 January 2020; Published: 3 February 2020

Abstract: Matrix transposition is a fundamental operation, but it may present a very low and hardly predictable data cache hit ratio for large matrices. Safe (worst-case) hit ratio predictability is required in real-time systems. In this paper, we obtain the relations among the cache parameters that guarantee the ideal (predictable) data hit ratio assuming a Least-Recently-Used (LRU) data cache. Considering our analytical assessments, we compare a tiling matrix transposition to a cache oblivious algorithm, modified with *phantom padding* to improve its data hit ratio. Our results show that, with an adequate tile size, the tiling version results in an equal or better data hit ratio. We also analyze the energy consumption and execution time of matrix transposition on real hardware with pseudo-LRU (PLRU) caches. Our analytical hit/miss assessment enables the usage of a data cache for matrix transposition in real-time systems, since the number of misses in the worst case is bound. In general and high-performance computation, our analysis enables us to restrict the cache resources devoted to matrix transposition with no negative impact, in order to reduce both the energy consumption and the pollution to other computations.

Keywords: transposition; data cache memory; real-time; tiling; cache oblivious

1. Introduction

Matrix transposition is a fundamental operation in linear algebra, fast fourier transforms, and so forth, and has many applications in areas such as numerical analysis, image processing and graphics. There are many examples of applications developed at supercomputing centers that, in one way or another, are using the transposition of matrices as part of the solution to their problems. For instance, the Oak Ridge National Laboratory develops, maintains, tests and manages the SCALE [1] Code System that is a widely-used modeling and simulation suite for nuclear safety analysis and design. As a second example, the Sandia National Laboratory developed the Geotess [2], a model parameterization and software support system that implements the construction, population, storage and interrogations of data stored in 3D Earth models. Although transposition is a very simple problem, its basic implementation may present a very low data cache hit ratio for large matrices [3]. This is due to access to consecutive elements in the matrix (which are likely to fit in the same cache line and thus present temporal reuse) suffering from many data accesses between them.

In order to overcome this problem, well known code transformations can be applied. *Tiling* or *blocking* present in high-performance libraries (e.g., Intel MKL, NVIDIA cuBLAS), dividing the whole problem into small tiles fitting in cache [4]. Hence, tiles of the original matrix are completely transposed one after another, so that each tile remains cached while it is processed, avoiding capacity misses. Essentially, this transformation implies adding external loops to the original code, so that the global access sequence is not spread in memory but localized into the working tile. Although tiling effectively increases the data cache hit ratio, it has two main drawbacks. First, the added loops imply that more instructions are required to perform the matrix transposition, with their corresponding increase in execution time. Second, the size of the matrix to transpose and the specific cache configuration in the target system (number of sets and ways, cache line size, replacement policy, victim cache, prefetch, etc.) affect the hit ratio of the tiled code. These drawbacks have been addressed from the compiler optimization perspective, trying to minimize the number of cache misses in the generated binary code [5]. In the case of real-time systems, these drawbacks are critical, since the *worst-case execution time* (WCET) must be known at design-time [6]. Hence, if hits and misses cannot be accurately predicted in advance, the real-time system will be designed assuming an overestimated behaviour, leading to a waste of hardware resources and an increment of its energy consumption. Moreover, real-time systems require a *safe* prediction, so techniques that do not guarantee the worst-case result are not applicable [7]. Also, these drawbacks may be important in cloud computing, where the bare metal target cache may be hidden and may change.

Alternatively, a *cache oblivious* transposition algorithm could be used. Essentially, cache oblivious algorithms could be seen as recursive versions of tiling algorithms, so that the optimal cache performance is reached by the specific nature of recursion. That is, a cache oblivious algorithm does not require any parameter based on an explicit knowledge the cache configuration. Focusing on matrix transposition, this means that it does not require the tile size parameter (mandatory in tiling), but each recursion divides the matrix to transpose into several smaller matrices to transpose until reaching a 2×2 size. As Tsifakis et al. conclude [8], "predicting a priori how the cache oblivious algorithm will perform is non-trivial", so previous drawbacks are still present. However, it avoids adding the tile size as an extra parameter to consider.

Independently of the implementation of matrix transposition, to the best of our knowledge no prior work provides an analytical hit/miss assessment of this problem. Without such analytics, performance trends depending on specific cache parameters can be studied, but these parameters cannot be accurately tuned and performance cannot be predicted with precision.

In this paper, we study the in-place matrix transposition theoretically, and how the cache parameters (number of sets, ways, and line size) in a Least-Recently-Used (LRU) cache, and the padding applied to the matrix affect the data hit ratio of the tiling version. Our findings are validated by means of simulations considering a wide range of parameters. We also compare our results to those of a cache oblivious implementation, enhanced with a new padding technique we term "phantom padding". Finally, we analyze the performance of matrix transposition on real hardware (with specific cache configurations) to verify that it complies with our theoretical predictions. Our contributions can be summarized as follows:

- Theoretical expressions to estimate the ideal data cache behaviour (compulsory misses) of matrix transposition, independently of the cache configuration and the matrix transposition algorithm.
- Theoretical expressions for an optimal LRU data cache configuration of the tiling version of matrix transposition.
- Validation of previous expressions and comparison to cache oblivious by means of simulations.
- Experimental results on real hardware and comparison to pseudo-LRU (PLRU).

Furthermore, our findings can be directly applied to current matrix transposition software-hardware with the following immediate benefits:

- No more than two ways of data caching and a few sets are required in general for matrix transposition, so the remaining ways in a set-associative data cache could be powered off, with its corresponding energy savings and no negative impact.
- Contemporary partitionable last-level caches could offer just two ways to the matrix transposition process, avoiding unnecessary pollution to other computations with no negative impact.

- Hit-miss results can be easily predicted for the tiling version of matrix transposition (mandatory for real-time applications) in an LRU cache and, under specific cache configurations, in a PLRU cache.

The rest of this paper is organized as follows: Section 2 outlines the related work on matrix transposition. Section 3 studies ideal data hit ratios for a cache with unlimited capacity and independently of the transposing algorithm. Next, in Section 4 we analyze the LRU data cache requirements to reach the previous ideal hit ratio for a tiling version of matrix transposition. In Section 5, previous requirements are refined for a matrix with extra padding. Our results are presented in Section 6, including the validation of previous analytical assessments, their comparison to a cache oblivious algorithm with phantom padding, and experiments on real hardware. Finally, Section 7 presents our conclusions.

2. Related Work

In this section, we explore some of the works that analyze the cache hit/miss behaviour of several matrix transposition algorithms. Due to the simplicity of the algorithm and the intensive use of memory, the transpose algorithm is in the crosshairs of several works, which try to analyze and improve the usage of the memory hierarchy.

Two alternatives have been explored in order to manage a cache conscious access pattern of the transposition algorithm: block tiling and recursive division of the data domain. One of the main differences among these two alternatives is about how much about the underlying cache memory has to be exposed to the algorithm. *Block tiling* requires adjusting the size of the tiles to the size of the caches, meanwhile *recursive subdivision* fragments the data domain into small pieces, that will certainty fit into the cache. This last approach does not require knowing details about the cache, so it is said to be cache oblivious. Both approaches look for exposure to the memory hierarchy a data memory access pattern that exploits the reuse (spatial and temporal) in the underlying cache [3,4,8–12]. Next, we present the differences between the related works and ours.

Cache-Efficient Matrix Transposition Reference [3] describes several matrix transposition algorithms and compares their performance using both simulation and real execution on a Sun UltraSPARC II based system.By simulation it is shown that, while the cache oblivious transposition algorithm has the smallest number of cache misses for small matrix dimensions, for large dimensions it is actually the worst. Moreover, the reported execution times show that, in most cases, the cache oblivious algorithm is significantly slower than the other transposition algorithms. They suggest that the lack of associativity is the reason why it does not perform as expected. In comparison to our approach, they limit their analysis to an experimental evaluation without providing analytical hit/miss assessment of the different algorithms.

Cache Oblivious Matrix Transposition: Simulation and Experiment Reference [8] explores further the cache oblivious matrix transposition algorithm with the aim of rationalizing the results of Chatterjee and Sen [3]. They study its performance, with respect to cache misses, both with simulation and hardware counters on two fundamentally different Sun UltraSPARC systems. As in our work, they compare tiling and oblivious algorithms but they focus on the oblivious behaviour. However, they test a single specific cache configuration and tile size, varying only the matrix size. Their results show that the cache miss characteristics of the algorithm present a rather structured pattern, which depends on the cache configuration and the matrix dimension. However, they do not conclude when the cache oblivious algorithm will perform well or poorly, but only that increasing the cache ways tends to improve the hit ratio. In our work, we analytically assess this observation.

The Cache Performance and Optimizations of Blocked Algorithms Reference [4] focuses on optimizing the cache performance via blocking (tiling). Their approach is to first discover the behaviour of caches under tiling, and then to improve its performance via software and/or hardware techniques. They analyze the matrix multiplication tiled algorithm and conclude that the performance of the cache is highly dependent on the problem size and the tile size. They report a high sensitivity of the miss rates to the size of the input matrix. However, as we will show in our proposal, a careful tuning of the tiled transposition algorithm gives the programmer many more degrees of freedom.

An Experimental Comparison of Cache-oblivious and Cache-conscious Programs Reference [10] compares experimentally cache-oblivious and cache-conscious (tiling transformation applied) programs for matrix multiplication and matrix transposition. They claim there is an overhead that cache-oblivious programs pay for the ability to adapt automatically to the memory hierarchy. They conclude that even highly optimized cache-oblivious programs perform significantly worse than the corresponding cache-conscious programs. However, unlike us, due to experimental limitations they did not analyze a wide range of cache configurations nor a wide range of tile sizes.

As a general consideration of the comparison with these related works, in their implementations and experimentation carried out, many of them use a matrix size multiple of the cache line size and tile size which eases the analysis [3,4,8], and others consider padding as a technique to avoid this limitation but they do not analytically assess its effects [10]. In any case, none have considered cache set collisions in the case of consecutive rows of the same column (column major access in transposition). We address this issue with additional padding, the *row-shift padding* [13–15]. A further problem arises for the cache oblivious matrix transposition algorithm, which benefits from the application of a *phantom padding* (to the best of our knowledge not yet published), which is also analyzed in this paper.

3. Ideal Data Cache Hit Ratios in Matrix Transposition

Data cache hit ratio is the percentage of data accesses that result in cache hits. So, if we consider an unlimited capacity cache, we can call its hit rate "ideal". More specifically, let us consider a cache with an unlimited number of lines, initially empty, and a line size able to hold L elements of the matrix to transpose ($1 \leq L \in \mathbb{N}$). Therefore, the ideal hit ratio only accounts for compulsory misses and not for capacity misses, conflict misses or the transposition algorithm. Nevertheless, Algorithm 1, implementing a straightforward matrix transposition, could be used as a reference.

Algorithm 1 TransposeMatrix($Matrix$, N): Transposes a $N \times N$ matrix.

```
1: for $index1 \leftarrow 1$ to $N$ do                                          # N iterations
2:     for $index2 \leftarrow index1 + 1$ to $N$ do                             # N iterations
3:         $temp \leftarrow Matrix_{index1,index2}$                             # Memory load
4:         $Matrix_{index1,index2} \leftarrow Matrix_{index2,index1}$          # Memory load and store
5:         $Matrix_{index2,index1} \leftarrow temp$                             # Memory store
6:     end for
7: end for
```

Modeling ideal cases is useful for setting upper bounds once a given cache configuration, element-to-line mapping or algorithm has been chosen. We will start considering that only the matrix elements requiring transposition account for cache line misses, thus excluding the elements of the diagonal and producing a simple upper bound for any ideal hit ratio. Then we will show ideal hit ratios including the effects of padding.

We consider a $N \times N$ matrix to transpose, with elements stored in row major order, aligned to the cache line size. All results of this paper are also valid for elements stored in column major order, but we discuss just the row major order for the sake of simplicity. Also, we assume that the resulting matrix overwrites the input matrix, and no other memory structures are used, apart from registers.

3.1. Bound of the Ideal Hit Ratio with a Cache Line Holding L Elements

Under previous considerations, an upper bound for the ideal hit ratio can be easily calculated. All elements except diagonal elements ($N^2 - N$) are accessed for both load and store, so there are $2(N^2 - N)$ accesses in total. Assuming these $N^2 - N$ elements fit perfectly in cache lines, there are $(N^2 - N)/L$ compulsory misses. Division by L implicitly means that diagonal elements do not share their cache line with other elements. Although this assumption is not realistic (it would complicate any indexed access), it provides the following upper bound for the ideal hit ratio:

$$1 - \frac{(N^2 - N)/L}{2(N^2 - N)} = 1 - \frac{1}{2L} \tag{1}$$

Now, in order to calculate other ideal data hit ratios, the relationship between matrix size and cache line size must be considered.

3.2. Ideal Hit Ratio in a Matrix with Row Size Multiple of Cache Line Size ($r = 0$)

We assume that the $N \times N$ matrix to transpose is stored in memory in a row major order and aligned to the cache line size, including the diagonal elements. Also, we assume a data cache line holding L consecutive row elements, with N multiple of L. Using the *modulo* notation, $r = (N \mod L) = 0$. See Figure 1a.

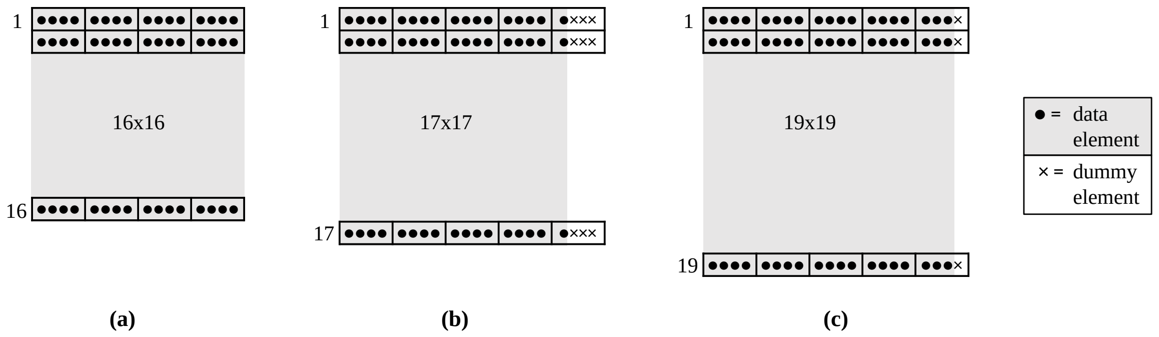


Figure 1. Different memory mappings when applying padding to a $N \times N$ matrix with cache lines holding L elements: (**a**) $r = (N \bmod L) = 0$, (**b**) $r = 1$, (**c**) $r = 3$.

The total number of accesses is the same as in Equation (1). The number of compulsory misses in this case is $(N/L)N$, that is, the number of cache lines required to traverse a row, times the number of rows. Hence, the ideal hit ratio is:

$$1 - \frac{N^2/L}{2(N^2 - N)} = 1 - \frac{1}{2L} - \frac{1}{2L(N-1)}. \tag{2}$$

3.3. Ideal Hit Ratio in a Matrix with ($r = 1$) Padding

When the number of elements in a matrix row (N) is not multiple of those fitting in a cache line (L), performance decreases. A well-known data transformation is padding, where each row in the matrix is extended with $L - r$ dummy elements to fill a cache line. Since the matrix is stored at a cache line aligned address, this allows the first element of each row to always start a new cache line. Therefore, for $r > 0$, each row produces a number of compulsory misses equal to $(N \text{ div } L) + 1 = N/L - r/L + 1$. In this section we assume $r = (N \mod L) = 1$, so each row in the matrix is padded with $L - 1$ elements. As a special case for $r = 1$, note that the last cache line of the matrix only keeps the last diagonal element, so it is never accessed. See Figure 1b. The ideal hit ratio with $r = 1$ is:

$$1 - \frac{(N/L - 1/L + 1)N - 1}{2(N^2 - N)} = 1 - \frac{1}{2L} - \frac{1}{2N}. \tag{3}$$

3.4. Ideal Hit Ratio in a Matrix with ($r > 1$) Padding

For the remaining cases of padding, $1 < r = (N \mod L) < L$ ($r \in \mathbb{N}$), see Figure 1c, the ideal hit ratio is:

$$1 - \frac{(N/L - r/L + 1)N}{2(N^2 - N)} = 1 - \frac{1}{2L} - \frac{1 + L - r}{2L(N-1)} \tag{4}$$

Figure 2 shows the ideal hit ratios for $L = 4$ elements when increasing the matrix size. Each line marks the corresponding r for each N, namely, $r = 0$ corresponds to Ns multiple of L (Equation (2)), $r = 1$ depicts Equation (3), and $r = 2$ and $r = 3$ represent Equation (4). All these lines tend to the data hit ratio asymptote $1 - \frac{1}{2L}$ (Equation (1)).

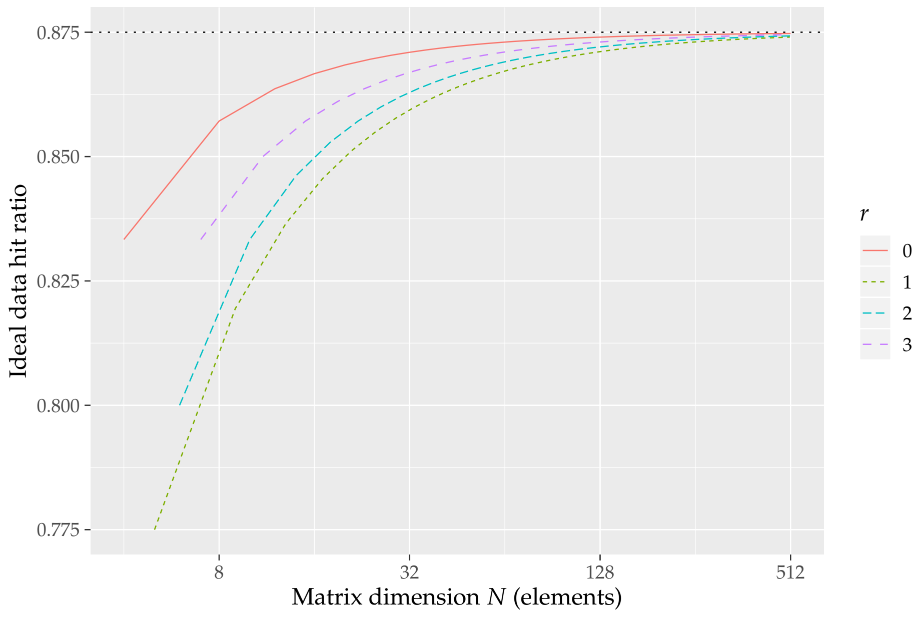


Figure 2. Ideal data hit ratios for $L = 4$ elements with different r values. Black dotted line shows the upper bound hit ratio of Equation (1).

4. Required LRU Cache Configuration to Achieve Ideal Hit Ratios

The previous section formulates the ideal data hit ratios for matrix transposition. In this section we analyze how to reach them by a set-associative LRU cache. We assume that the matrix is transposed by working on tiles. This corresponds to the tiling algorithm of matrix transposition (Algorithm 2). It divides the original matrix into small tiles fitting in the cache [4]. Hence, tiles are completely transposed one after another, so that each tile remains cached while it is processed, avoiding capacity misses. Essentially, this transformation implies adding external loops to the original code, so that the global access sequence is not spread, but localized into the working tile. We focus on the relation between the matrix size ($N \times N$ elements, row major order), the tile size ($T \times T$ elements) and the LRU cache configuration (S sets, W ways per set, and lines able to hold L elements). Excluding diagonal tiles, this implies working with pairs of tiles. One of them is traversed in row order, and the other one in column order. In each tile, all rows of T horizontal elements are swapped with the T vertical elements in the other tile. The ideal hit ratio is achieved when the available cache resources are enough and they are used effectively. That is, the size of the cache line L and the number of sets S must suffice for effectively caching the transposition of each pair of tiles. Otherwise, the cache associativity (W) must be large enough to avoid the possible conflict misses. For the tile traversed in row order a single cache line suffices, since when all elements in a cache line have been transposed, the cache line is no longer accessed. For the tile traversed in column order, all cache lines holding the elements of each column must remain cached until all the elements of these cache lines have been transposed, which requires either a minimum number of sets or a minimum associativity. Let us focus on the tile traversed in column order. Ideally, the number of sets required would be equal to the number of rows of each tile, that is, T sets. However, if several elements in a column are mapped to the same set, additional sets would be required. This depends on the relation (*greatest common divisor*) between the number of memory lines in a row ($\lceil N/L \rceil$) and the number of sets S. If the number of sets is enough, any direct-mapped cache ($W = 1$) would suffice to hold the required cache lines. Otherwise, the additional cache lines required must be supplied by an increased number of ways W (where each way consists of S cache lines). Hence, the number of ways required for the tile traversed in column order is:

$$W_{ColumnOrder} = \left\lceil \frac{T}{\left\lceil \frac{S}{\gcd(\lceil N/L \rceil, S)} \right\rceil} \right\rceil \tag{5}$$

Algorithm 2 Tiling($PaddedMatrix$, N, T): Transposes a $N \times (N + padding)$ padded matrix, stored in row major order, through $T \times T$ tiles [4].

```
1:  for $i \leftarrow 1$ to $N$ step $T$ do                         # triangular matrix traversal, by tiles
2:    for $j \leftarrow 1$ to $i$ step $T$ do
3:      for $index1 \leftarrow i$ to $\min(i + T - 1, N)$ step 1 do        # transpose non-diagonal tiles
4:        for $index2 \leftarrow j$ to $\min(j + T - 1, N)$ step 1 do
5:          $temp \leftarrow PaddedMatrix_{index1,index2}$
6:          $PaddedMatrix_{index1,index2} \leftarrow PaddedMatrix_{index2,index1}$
7:          $PaddedMatrix_{index2,index1} \leftarrow temp$
8:        end for
9:      end for
10:   end for
11:   for $index1 \leftarrow i$ to $\min(i + T - 2, N)$ step 1 do          # transpose diagonal tiles
12:     for $index2 \leftarrow index1 + 1$ to $\min(i + T - 1, N)$ step 1 do
13:       $temp \leftarrow PaddedMatrix_{index1,index2}$
14:       $PaddedMatrix_{index1,index2} \leftarrow PaddedMatrix_{index2,index1}$
15:       $PaddedMatrix_{index2,index1} \leftarrow temp$
16:     end for
17:   end for
18: end for
```

Remember that an additional cache line is required for the tile traversed in row order. Further, the specific ordering of the LRU replacement policy when processing *all* rows/columns in a tile may require another additional cache line. For instance, a fully associative cache ($S = 1$) requires 4 lines to complete the transposition of a tile with $T = L = 2$. So, two extra ways could be required (at most) to achieve the ideal hit ratio:

$$W = \left\lceil \frac{T}{\left\lceil \frac{S}{\gcd(\lceil N/L \rceil, S)} \right\rceil} \right\rceil + 2. \tag{6}$$

Since the previous equation takes into account many different situations, a more detailed analysis for each case is required, which can be refined with further padding.

5. Reducing the Requirements of Set-Associative LRU Caches with Further Padding

Previous requirements assume a matrix where each row has been padded to fill the cache line (Section 3.3). In this section we assume an extra padding in order to achieve the ideal hit ratios with fewer cache resources. If needed, we add to each row an additional *row-shift padding* equivalent to the cache line size L to ensure that the first elements in two consecutive rows are mapped to different sets. This is also a well-known padding transformation, intended to minimize the conflict misses [13–15]. So, there are no conflicts between the cache lines holding elements of the same column, as long as $S \geq T$ (gcd $= 1$ in Equation (6)).

We divide our analysis into three cases depending on the relation between the tile size and the cache line size, namely $T = L$, $T > L$, and $T < L$, with $T \geq 1$, $T \in \mathbb{N}$.

5.1. Bounds on W to Achieve Ideal Hit Ratios with $T = L$

Algorithm 2 transposes a matrix tile by tile. In every tile, T horizontal elements are swapped with T vertical ones. Figure 3 shows a schema of the elements to swap and the cache lines holding them, for a given tile. With $T = L$, padding (Section 3.3) ensures that all horizontal elements in the tile fit in a single cache line. Also, *row-shift padding* [13–15] ensures that cache lines holding elements of the same column in the tile are mapped to different cache sets. In the following analysis we consider the number of sets, distinguishing the cases $S \geq L$, $1 < S < L$ and $S = 1$.

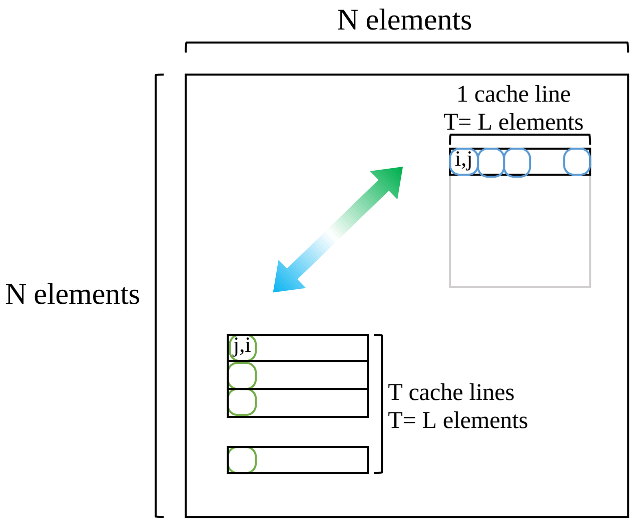


Figure 3. In-place transposition of a row-major stored $N \times N$ matrix assuming $T = L$.

5.1.1. Case $S \geq L$

Since $S \geq T$, *row-shift padding* ensures that the only possible conflict involves the cache line holding the horizontal elements and one of the cache lines holding a vertical element. This single potential conflict miss is avoided by any LRU cache with at least 2 ways. Figure 4 shows the placement of such cache lines in a set-associative cache with $S \geq L$. Therefore, the ideal hit ratio (Section 3) can be reached with an LRU cache with $T = L$, $S \geq L$, $W \geq 2$, independently of the matrix size. In other words, the cache associativity required with $T = L$ and $S \geq L$ to reach the ideal hit ratio is bounded by 2:

$$W = 2, \text{ with } T = L,\ S \geq L. \tag{7}$$

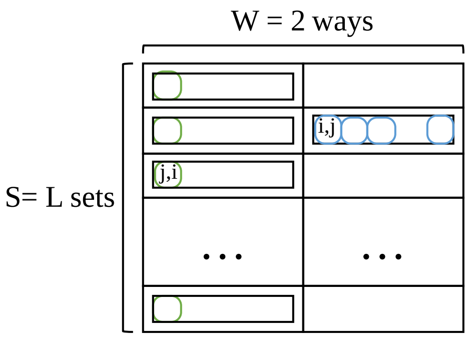


Figure 4. Set-associative cache holding a whole tile ($T = L$), with $S \geq L$ and $W = 2$.

5.1.2. Case $1 < S < L$

In general, caches have a rather high number of sets, so Equation (7) models the common case. For the sake of completeness, let us consider the $S < L$ case. With $S < L$, new conflicts may arise between the lines holding the vertical elements to transpose. Hence, in order to reach the ideal hit ratio further ways are required to absorb such conflicts. Specifically, $\lceil T/S \rceil$ ways are required to avoid conflicts between the vertical elements, plus the additional way to avoid conflicts with the cache line holding the horizontal elements to transpose. Figure 5 shows a cache representation for this case. With these conditions, the associativity required to reach the ideal hit ratio is bounded by:

$$W = \left\lceil \frac{T}{S} \right\rceil + 1, \text{ with } T = L,\ 1 < S < L. \tag{8}$$

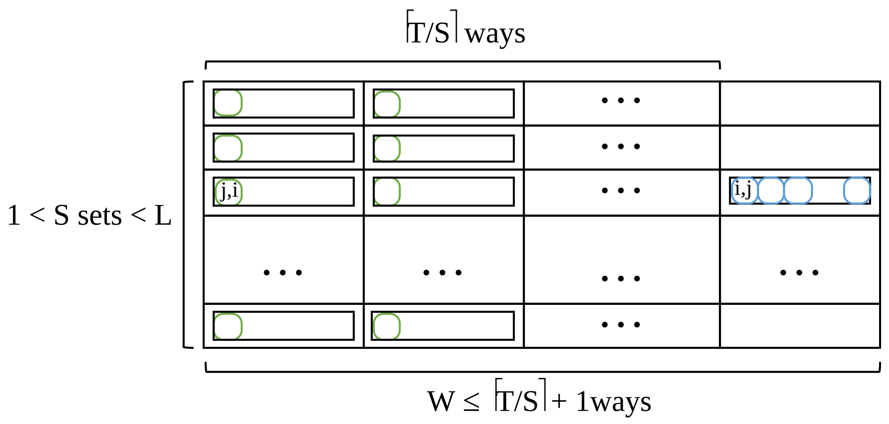


Figure 5. Set-associative cache holding a whole tile ($T = L$), with $1 < S < L$.

5.1.3. Case $S = 1$

Lastly, fully associative caches ($S = 1$) are a particular case, since associativity must be large enough to provide all the cache lines required to process each row/column in a tile, that is, $T + 1$ cache lines. However, the specific ordering of the LRU replacement policy when processing *all* the rows/columns in a tile, may require an additional cache line. Thus, the cache associativity required with $T = L$ and $S = 1$ to reach always the ideal hit ratio is bounded by:

$$W = T + 2, \text{ with } T = L,\ S = 1. \tag{9}$$

For instance, a fully associative cache requires 6 lines to complete the transposition of a tile with $T = L = 4$. Let us use this example to illustrate the correctness of Equation (9). Figure 6 shows the memory mapping of two tiles (4×4 elements) to be transposed in a $N \times N$ element matrix. The eight cache lines involved in the transposition are B_a to B_h. In each line, a distinct pair of indexes identifies each element to transpose. In order to conduct the transposition of these two tiles, Table 1 shows the sequence of memory access instructions arising during the tile transposition: loads (ld) and stores (st) of elements $E_{row,column}$. For each step, Table 1 shows a LRU stack of size 6 ($T + 2$, accordingly to Equation (9)). On top, the MRU position corresponds to the most recently used line, this is, the line holding the last load/store element accessed. On bottom, the LRU position holds the cache line least recently referenced. In a fully-associative cache with an LRU replacement policy, the line at the LRU position is the candidate for replacement in case of miss. Steps 1 to 4 represent the transposition of elements $E_{i,j}$ and $E_{j,i}$. They correspond to lines 5 to 7 in Algorithm 2. Steps 1 and 2 read the two elements in the original matrix from their corresponding cache line, B_a and B_e, respectively. Steps 3 and 4 store these elements to their transposed position in the matrix. For instance, the most recently used cache line in step 4 is B_e since the last instruction executed ("st $E_{j,i}$") stores to $E_{j,i}$ (in B_e) the content previously read ("ld $E_{i,j}$" in step 1) from index (i, j) (in B_a). At step 16, all elements in cache line B_a have been transposed with the corresponding elements of cache lines $B_{[e,f,g,h]}$. At this point, the LRU stack holds 5 cache lines, this is $T + 1$. Next (steps 17 to 32), elements in line B_b must be transposed with the corresponding elements of lines $B_{[e,f,g,h]}$. In order to assure that none of the cache lines B_e, B_f, B_g, B_h is evicted when cache line B_b comes in, we need an additional entry, this is $T + 2$.

At step 32, the least recently used cache line is B_a, which has not been used during the transposition of line B_b in steps 16 to 32. So, B_a can be evicted without loss at step 33, when we start the transposition of next cache line B_c.

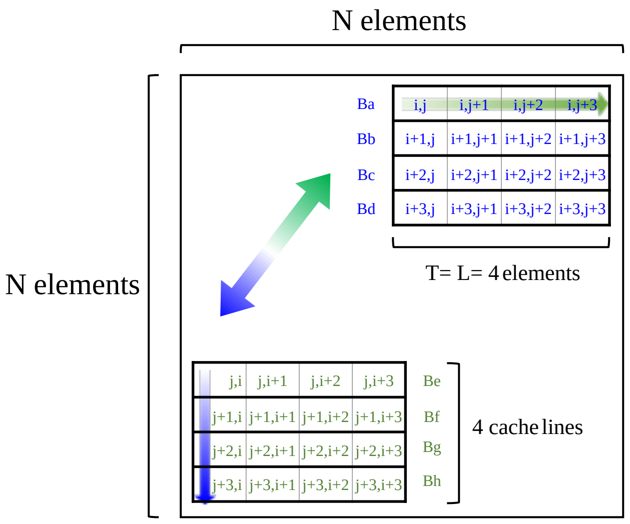


Figure 6. Memory mapping of the blocks $B_{i\in[a-h]}$ holding the elements to transpose for $T = L = 4$ elements.

5.2. Bound on W to Achieve Ideal Hit Ratios with $T > L$

Larger tiles require less instruction overhead in the outer loops of Algorithm 2, so it is interesting to establish the conditions to reach the ideal hit ratio with larger tile sizes ($T > L$). In such cases, $\left\lceil \frac{T/L}{S} \right\rceil$ additional cache lines are required to hold the horizontal elements to transpose. Depending on the relationship between the LRU cache parameters and the matrix size, the ideal hit ratio can be reached with a slightly different number of ways. For instance, matrices with N not multiple of 2 require one way less. In any case, the associativity required to reach the ideal data hit ratio for any S is bounded by:

$$W = \left\lceil \frac{T}{S} \right\rceil + \left\lceil \frac{T/L}{S} \right\rceil + 1, \text{ with } T > L. \tag{10}$$

5.3. Bound on W to Achieve Ideal Hit Ratios with $T < L$

Using a tile whose number of elements per row T is lower than those fitting in a cache line L is a very impractical assumption, since it implies that the processing of a $T \times T$ tile brings to cache more content than the strictly required to process the tile. Hence, in order to reach the ideal hit ratio, such unused content must not be evicted until it is effectively used. In other words, it is required a cache large enough to avoid capacity misses. Specifically, the number of cache lines ($S \times W$) must be larger than twice the number of columns in the matrix ($2N$). With such an unsuitable tile size, the number of ways required to reach the ideal hit ratio depends on the matrix size, bounded as follows:

$$W = \left\lceil \frac{2N}{S} \right\rceil + 1, \text{ with } T < L. \tag{11}$$

Table 1. Evolution of the LRU stack content during a tile transposition in a cache with $L = T = 4$ and $S = 1$. Each LRU stack snapshot is taken after the execution of ld and st instructions that access the $E_{row,column}$ elements, and it shows the ordered sequence of all the previously accessed B_i cache lines.

	Sequence of Memory Access Load (ld) and Store (st) Instructions								
	1 ld $E_{i,j}$	2 ld $E_{j,i}$	3 st $E_{i,j}$	4 st $E_{j,i}$	5 ld $E_{i,j+1}$	6 ld $E_{j+1,i}$	7 st $E_{i,j+1}$	8 st $E_{j+1,i}$	9 ld $E_{i,j+2}$
MRU	B_a	B_e	B_a	B_e	B_a	B_f	B_a	B_f	B_a
1		B_a	B_e	B_a	B_e	B_a	B_f	B_a	B_f
2						B_e	B_e	B_e	B_e
3									
4									
LRU									
	10 ld $E_{j+2,i}$	11 st $E_{i,j+2}$	12 st $E_{j+2,i}$	13 ld $E_{i,j+3}$	14 ld $E_{j+3,i}$	15 st $E_{i,j+3}$	16 st $E_{j+3,i}$	17 ld $E_{i+1,j}$	18 ld $E_{j,i+1}$
MRU	B_g	B_a	B_g	B_a	B_h	B_a	B_h	B_b	B_e
1	B_a	B_g	B_a	B_g	B_a	B_h	B_a	B_h	B_b
2	B_f	B_f	B_f	B_f	B_g	B_g	B_g	B_a	B_h
3	B_e	B_e	B_e	B_e	B_f	B_f	B_f	B_g	B_a
4					B_e	B_e	B_e	B_f	B_g
LRU								B_e	B_f
	19 st $E_{i+1,j}$	20 st $E_{j,i+1}$	21 ld $E_{i+1,j+1}$	22 ld $E_{j+1,i+1}$	23 st $E_{i+1,j+1}$	24 st $E_{j+1,i+1}$	25 ld $E_{i+1,j+2}$	26 ld $E_{j+2,i+1}$	27 st $E_{i+1,j+2}$
MRU	B_b	B_e	B_b	B_f	B_b	B_f	B_b	B_g	B_b
1	B_e	B_b	B_e	B_b	B_f	B_b	B_f	B_b	B_g
2	B_h	B_h	B_h	B_e	B_e	B_e	B_e	B_f	B_f
3	B_a	B_a	B_a	B_h	B_h	B_h	B_h	B_e	B_e
4	B_g	B_g	B_g	B_a	B_a	B_a	B_a	B_h	B_h
LRU	B_f	B_f	B_f	B_g	B_g	B_g	B_g	B_a	B_a
	28 st $E_{j+2,i+1}$	29 ld $E_{i+1,j+3}$	30 ld $E_{j+3,i+1}$	31 st $E_{i+1,j+3}$	32 st $E_{j+3,i+1}$	33 ld $E_{i+2,j}$			
MRU	B_g	B_b	B_h	B_b	B_h	B_c			
1	B_b	B_g	B_b	B_h	B_b	B_h			
2	B_f	B_f	B_g	B_g	B_g	B_b			
3	B_e	B_e	B_f	B_f	B_f	B_g			
4	B_h	B_h	B_e	B_e	B_e	B_f			
LRU	B_a	B_a	B_a	B_a	B_a	B_e			
evicted						B_a			

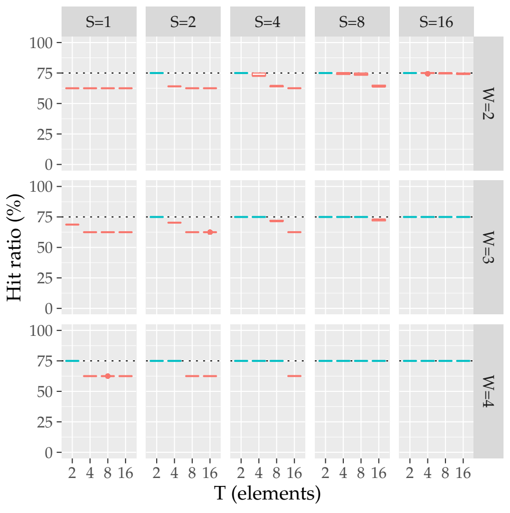
6. Experiments

In this section we validate previous analytical expressions by means of extensive simulations. We compute data cache hit ratios on different cache configurations for a wide set of matrix sizes. Further, the obtained results are compared with those provided by a cache-oblivious algorithm, demonstrating that if the tile size $T \times T$ is set to the cache line size ($T = L$), tiling always performs equal to or better than cache-oblivious. Also, we perform multiple experiments on real hardware in order to study the completion times (and not just the data cache hit ratio). To close this section, we use the results obtained to model the energy consumption of the cache hierarchies of each of the studied systems. This allows us to assess the energy waste due to the behaviour of the replacement activity in the cache hierarchy.

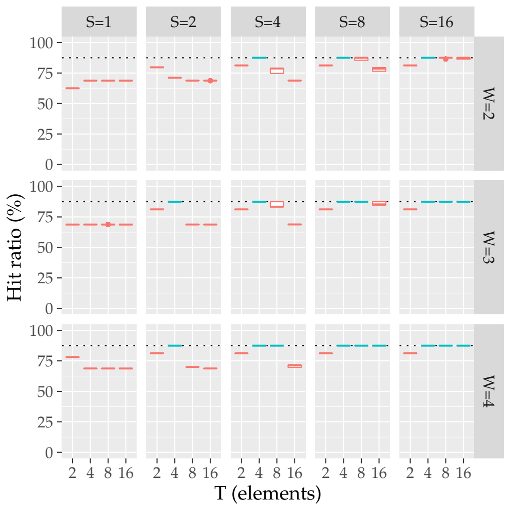
6.1. Data Cache Hit Ratio in Tiled Matrix Transposition

In this section we validate the reachability of the ideal data hit ratios (Section 3) for the tiled matrix transposition. An LRU data cache is assumed and the matrix to transpose is padded as described in Section 5. Figure 7 shows several plots corresponding to different line sizes L (in elements). All figures show a boxplot for each combination of number of sets (S), ways (W), and tile sizes (T). Each boxplot extends vertically and summarizes the data hit ratio for matrices from 1024×1024 to

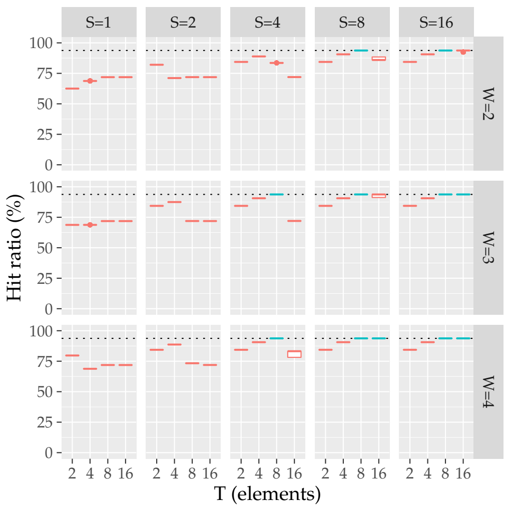
2048 × 2048 elements, that is, 1025 experiments. Most cases show just the median, appearing then as flat marks, which means that the data hit ratio of all matrices are equal or presents inappreciable differences. The horizontal dotted line represents the hit ratio asymptote $1 - \frac{1}{2L}$ (Equation (1)). Boxplot colours show whether the ideal hit ratio (Section 3) was reached for every matrix size (green) or not (orange). As can be seen, the ideal hit ratio is always reached when the *x*-axis (T) matches the line size in the columns ($T = L$), depending on the number sets S (Equations (7)–(9)). When ideal results are reached with a suboptimal tile size ($T > L$, Equation (10)), an optimal tile size ($T = L$) also reaches the ideal hit ratio. Note also that the properties of LRU policy guarantee that, if the ideal hit ratio is reached under certain cache configuration, it is also reached when increasing the number of sets and/or ways. Only if the cache is too small or its parameters are not tuned, the ideal hit ratio might be unreachable. Fortunately, such cases are not realistic, and existing data caches are far larger than those depicted in Figure 7. Indeed, the largest cache tested ($W = 4$, $S = 16$) with a line size of 64 bytes has a total size of just 4 KiB.



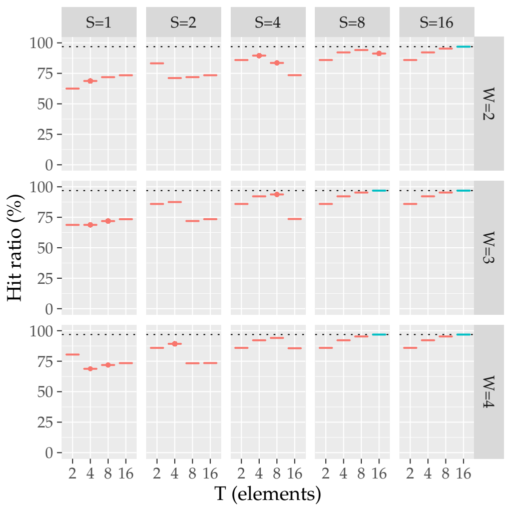
(a) $L = 2$



(b) $L = 4$



(c) $L = 8$



(d) $L = 16$

Figure 7. Plots (**a**–**d**) show data cache hit ratios varying the number of elements per line ($L = 2, 4, 8, 16$ elements) for different cache configurations with S sets, W ways, and tile sizes of $T \times T$ elements. Each boxplot gathers the results of all matrices from 1024 × 1024 to 2048 × 2048 elements. Boxplot colour shows whether the ideal hit ratio is always reached (green) or not (orange).

6.2. Tiling vs. Oblivious Data Hit Ratio

Previous results show that, with a correct tile size, the ideal hit ratio is reached by very small caches. Instead of a cache-conscious tiling algorithm, a cache-oblivious algorithm could be used. In this section we evaluate the data cache requirements to achieve the ideal hit ratio with a cache-oblivious algorithm [9,12], and compare it to our tiling results. A cache-oblivious algorithm could be seen as a tiling algorithm where the matrix to process is divided recursively until reaching, in matrix transposition, a tile size of 2×2 elements. Hence, the processing of each tile is likely to fit in cache, and the processing order imposed by recursion is a variation of the *Z-order*, which preserves locality [16]. Thus, the application of such an algorithm does not require a knowledge of the cache configuration. Nevertheless, its performance does depend on the cache parameters, and predicting when it will perform well or poorly is not trivial [8].

Since the oblivious algorithm works by recursively dividing the matrix to transpose, it presents problems when the matrix dimension (N) is not power of 2. This effect can be seen in Figure 8, where only matrices whose dimension is power of two achieve the ideal hit ratio. In order to overcome such drawback, we propose the application of a *phantom padding*. It does not modify the memory mapping of the matrix to transpose, but forces the oblivious algorithm to work as if the matrix dimension is power of 2. Then, swapping of elements is triggered only if they are real elements, that is, swapping of phantom elements is not applied. Such transformation does not affect the number of data accesses but just ensures the regular access sequence expected by the oblivious algorithm. To the best of our knowledge, *phantom padding* has not yet been published.

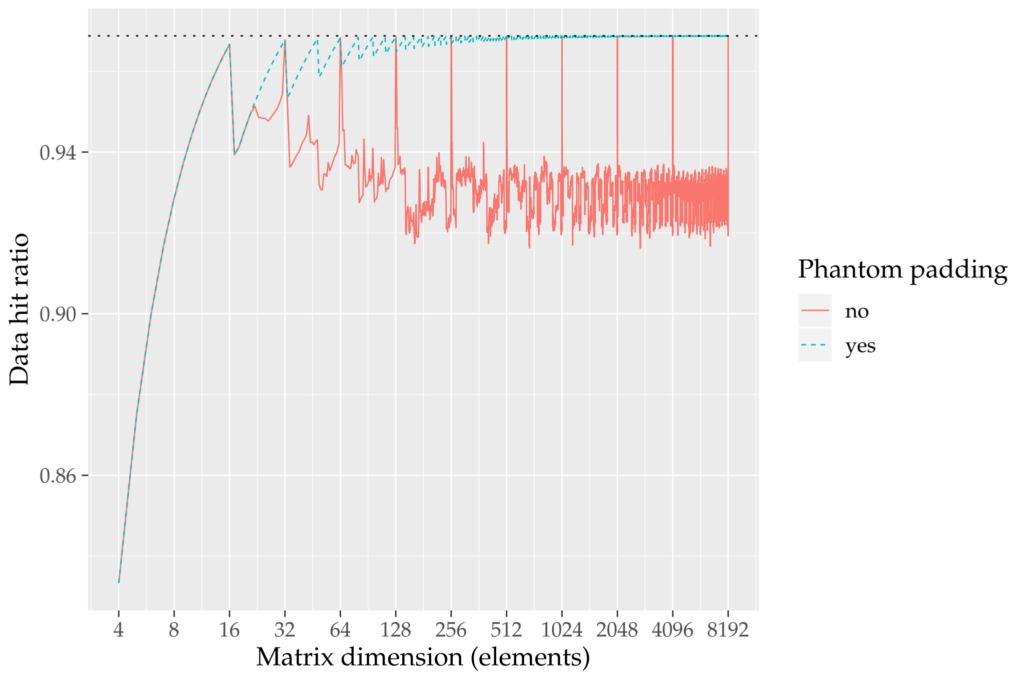


Figure 8. Data cache hit ratios without and with phantom padding (Algorithms 3 and 4) for the cache oblivious matrix transposition. The data cache is set to $L = 16, S = 16$ and $W = 2$. Matrix sizes vary from 4×4 to 8192×8192. The ideal data hit ratio (Equations (2)–(4)) matches that of phantom padding. Black dotted line shows the upper bound hit ratio of Equation (1).

Algorithms 3 and 4 show the modified cache-oblivious matrix transposition algorithms, based on the original proposals [9,12]. Essentially, such algorithms assume an N parameter being power of 2 and the actual matrix to transpose specified by the index parameters. So, actual swapping is performed only when the corresponding index is less than N. That is, we introduce the condition at line 6 in Algorithm 3 and the condition at line 8 in Algorithm 4.

Algorithm 3 Transpose($PaddedMatrix$, N, $index1 = 1$, $index2 = N$): transposes a *phantom-padded* matrix, stored in row major order, recursively.

```
1:  **if** $index2 - index1 \leq 2$ **then**
2:      $PaddedMatrix_{index1,index1+1} \leftrightarrow PaddedMatrix_{index1+1,index1}$
3:  **else**
4:      $indexhalf \leftarrow (index1 + index2)/2;$
5:      $Transpose(PaddedMatrix, N, index1, indexhalf);$
6:      **if** $indexhalf < N$ **then**
7:          $Transpose(PaddedMatrix, N, indexhalf, index2);$
8:          $TransposeSwap(PaddedMatrix, N, indexhalf, index1, index2, indexhalf);$
9:      **end if**
10: **end if**
```

Algorithm 4 TransposeSwap($PaddedMatrix$, N, rs, cs, re, ce).

```
1:  **if** $((re - rs) \leq 2$ **and** $(ce - cs) \leq 2)$ **then**
2:      **for** $index1 \leftarrow rs$ **to** $re - 1$ **do**
3:          **for** $index2 \leftarrow cs$ **to** $ce - 1$ **do**
4:              $PaddedMatrix_{index1,index2} \leftrightarrow PaddedMatrix_{index2,index1}$
5:          **end for**
6:      **end for**
7:  **else**
8:      **if** $rs < N$ **then**
9:          $rhalf \leftarrow (rs + re)/2;$
10:         $chalf \leftarrow (cs + ce)/2;$
11:         $TransposeSwap(PaddedMatrix, N, rs, cs, rhalf, chalf);$
12:         $TransposeSwap(PaddedMatrix, N, rhalf, cs, re, chalf);$
13:         $TransposeSwap(PaddedMatrix, N, rs, chalf, rhalf, ce);$
14:         $TransposeSwap(PaddedMatrix, N, rhalf, chalf, re, ce);$
15:     **end if**
16: **end if**
```

Assuming such improved behaviour, we have repeated the experiments of Section 6.1 for the oblivious algorithm. Figure 9 summarizes our results for tiling (Figure 7), and those for oblivious. It shows the minimum number of cache ways W required to reach the ideal hit ratio for each combination of cache line size L and number of cache sets S, that is, the lower the better, with tiling results on the left side and oblivious on the right one. Only adequate tile sizes ($T = L$) are shown. Inadequate tiling configurations ($T \neq L$) present worse results (not shown). Bars with green colour mark the minimum number of sets required to achieve the ideal hit ratio. It can be seen that oblivious requires the same number of ways as tiling on caches with a sufficient number of sets ($S \geq L$). However, on caches with a smaller number of sets (including fully-associative caches), oblivious requires more ways than tiling in order to reach the ideal hit ratio. This is due to oblivious does not stop recursion when reaching the most adequate (cache conscious) tile size, but digs until reaching the 2×2 tiles. Hence, the larger the cache line, the more unnecessary content it brings from memory. Such unnecessary content (which will be required for subsequent tiles) must remain cached to achieve the ideal data hit ratio. So, fixing the number of sets, the number of ways required grows.

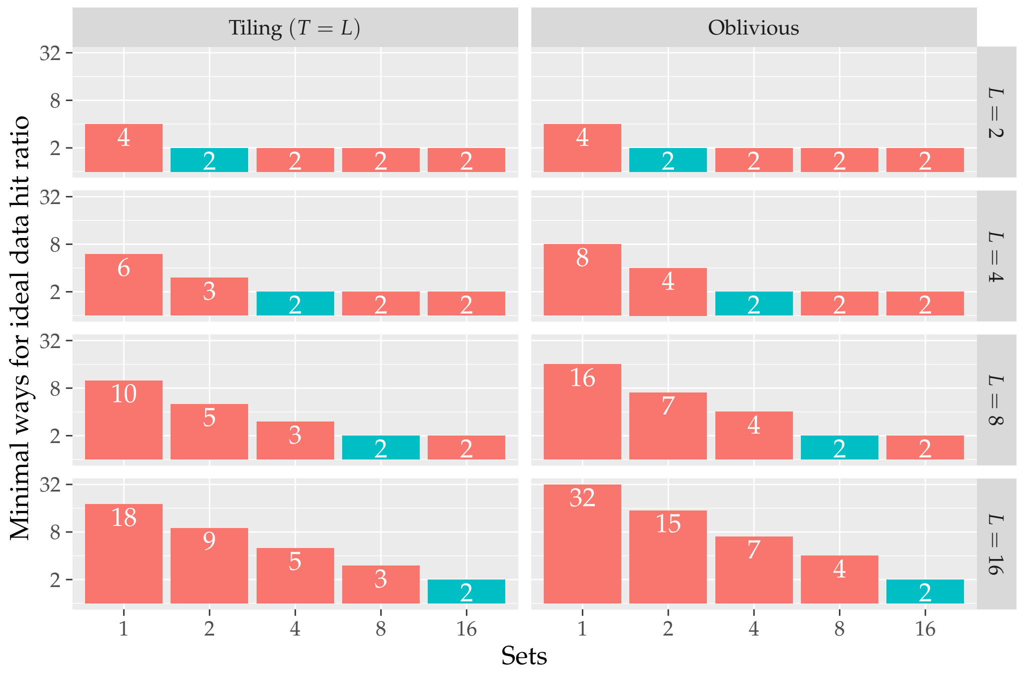


Figure 9. Minimum number of ways (W) required to achieve the ideal data hit ratio, depending on the number of cache sets S and line size L, for matrix transposition with a $T \times T$ tiling algorithm ($T = L$) and for a phantom-padded cache-oblivious algorithm (the lower the better).

Common data caches have a relatively large number of sets ($S \geq L$), so in general both cache-conscious and cache-oblivious matrix transposition should reach the ideal data hit ratio. Nevertheless, note that caches in current processors may be shared by different hardware threads at the same time, so accessing them as efficiently as possible is still important. Previous results show that tiling achieves the ideal data hit ratio with fewer resources than those required by oblivious. Also note that recursive codes such as the oblivious algorithm are usually slower than iterative codes, due to the function call and stack processing overheads.

6.3. PLRU and Execution Time on Real Platforms

Previous results assume an LRU set-associative data cache. Since building an efficient LRU policy for a large cache is costly, most caches in commercial processors use the PLRU policy, which offers a similar behavior and is simpler to build [17]. In this section, we first compare the LRU data hit ratio to that of PLRU by means of simulations. Figure 10 shows the resulting data hit ratios for a 4096×4096 8B-elements matrix transposition on a common L1 data cache configuration (64 sets, 8 ways, and 64 B per line, with a total size of 32 KiB). With such a line size, each cache line holds 8 elements of the matrix. Experiments vary the tile dimension from $T = 2$ to $T = 512$. The shaded area shows the tile sizes that achieve the ideal data hit ratio with LRU for tiles multiple of the line size ($L = 8$). This can be clearly seen between 8 and 16, where the data hit ratio decreases. The 8×8 tile corresponds to $T = L$ (ideal hit ratio with minimum cache resources), and larger tiles up to 256×256 reach the ideal hit ratio by using more cache sets/ways (Equation (10)). As can be seen, both LRU and PLRU present an almost identical data hit ratio. It decreases for LRU outside the shaded area as expected, and PLRU also presents this behavior. The only appreciable differences, although minimal, occur around $T = 256$, that is, the bound of Equation (10). Nevertheless, note that in areas such as real-time systems predictability is also very important. Such predictability is guaranteed in LRU by our findings, whereas PLRU should be avoided for these systems, as stated by other studies [6,18,19].

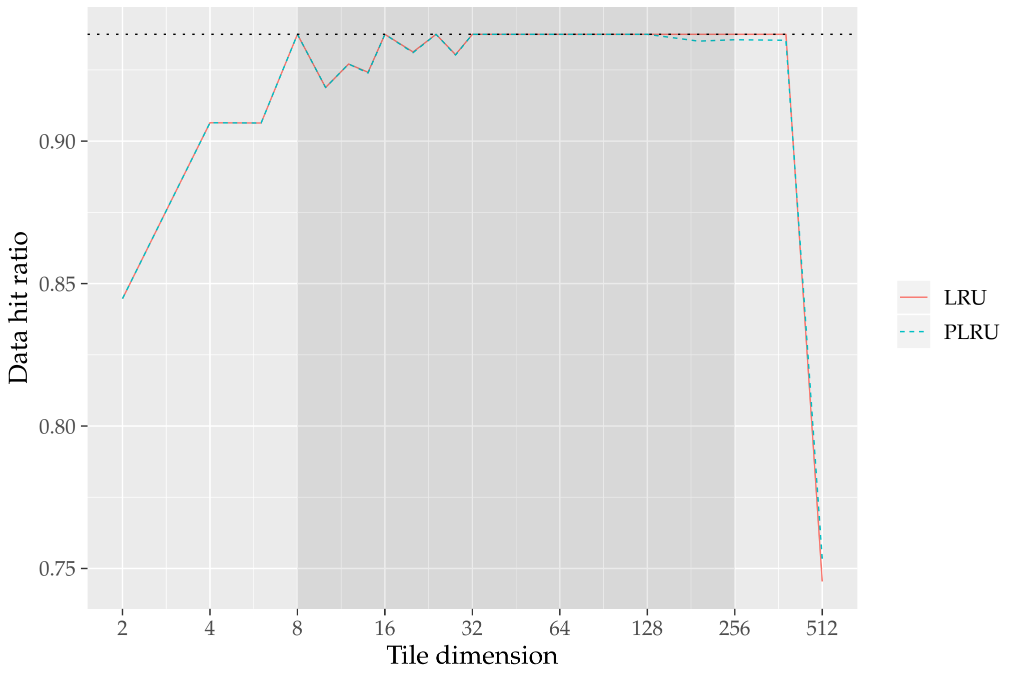


Figure 10. Data hit ratios for 4096 × 4096 matrix transposition (the higher the better) of Least-recently-Used (LRU) and pseudo-LRU (PLRU) on a common L1 data cache configuration (64 sets, 8 ways, and 64 B per line, with a total size of 32 KiB). Black dotted line shows the upper bound hit ratio of Equation (1).

Although the data hit ratio presented above is probably the most determinant factor regarding performance, other elements such as the number of executed instructions (dependent on the tile size) are also important. In order to consider these and other factors (e.g., hardware data prefetching), in this section we also measure execution times. Experiments have been carried on several target machines, namely Intel-Xeon-L5410 2.33 GHz, Intel-i7-4810MQ 2.80 GHz, Intel-Core-2-Quad-Q9550 2.83 GHz, and Intel-i7-2640M 2.80 GHz. All of them have L1 instruction and data caches with PLRU replacement, both with 64 sets, 8 ways, and 64 B per line, with a total size of 32 KiB, as the one simulated in Figure 10. Algorithm 2 has been coded (source code available at https://webdiis.unizar.es/gaz/repositories/tiling-matrix-transposition) in C and, for each machine, a different binary has been generated using its particular available compiler (gcc-4.7.0 for i7-4810M and Core-2-Quad-Q9550, Intel C++ Composer XE 2013 for Xeon-L5410, and gcc-4.9.2 for i7-2640), always with optimization level 3.

Figure 11 shows the lowest execution times for a 4096 × 4096 matrix transposition, varying the number of elements per tile. Each reported time is the lowest one among 400 repetitions, representing the fastest execution, that is, the execution closest to an isolated matrix transposition without external interferences. The shaded area shows the tile configurations that achieve the ideal data hit ratio with LRU for tiles multiple of the line size, as above. Times should grow for LRU outside the shaded area, and indeed it can be seen that PLRU presents this behaviour. Except for the 2QuadQ9550, the execution time for large tiles grows, but very slightly. This is due to the data prefetcher, which recognizes the data access pattern and avoids time penalties for the other three platforms. For the left area, tiles are too small for the data prefetcher to recognize the access pattern, so the required execution times for these tiles are rather large. Sizes multiple of 8 work with whole lines, so they perform better than surrounding sizes in general. This can be clearly seen between 8 and 16, where all plots show an increased execution time. Lastly, the larger the tile size, the less executed instructions. For each tile, its index bounds must be calculated, also ensuring that border tiles do not swap data outside the matrix. That is, the smaller tile sizes inside the shaded area have the most efficient usage of the cache, but those on the right execute less instructions. Hence, there is a trade-off between cache resources and executed instructions, and the observed trend regarding the required execution time may be different depending on the target machine.

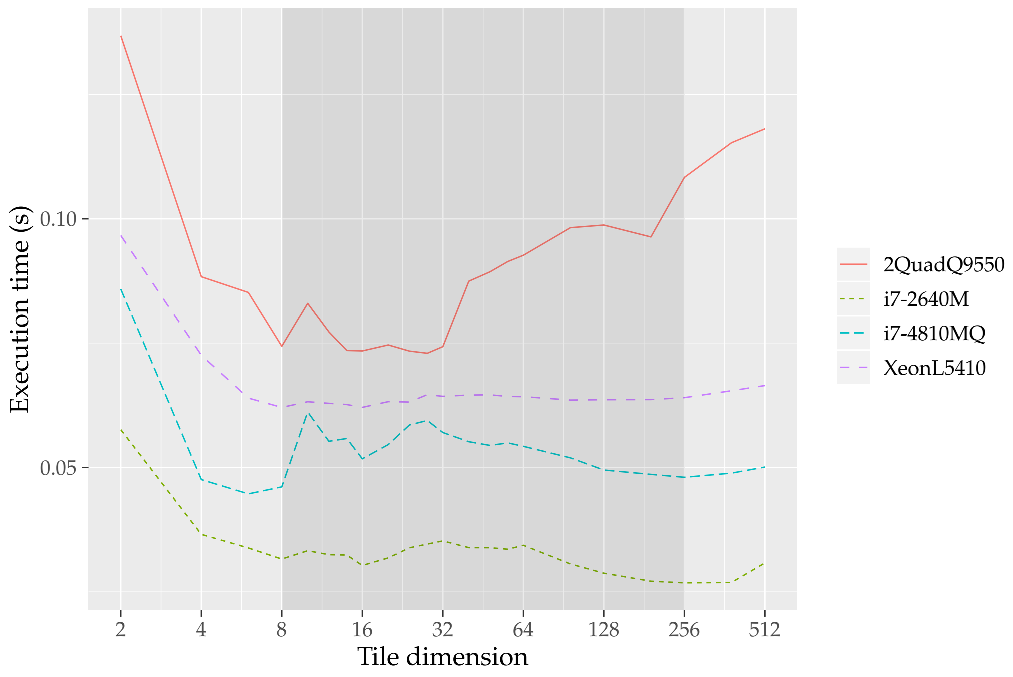


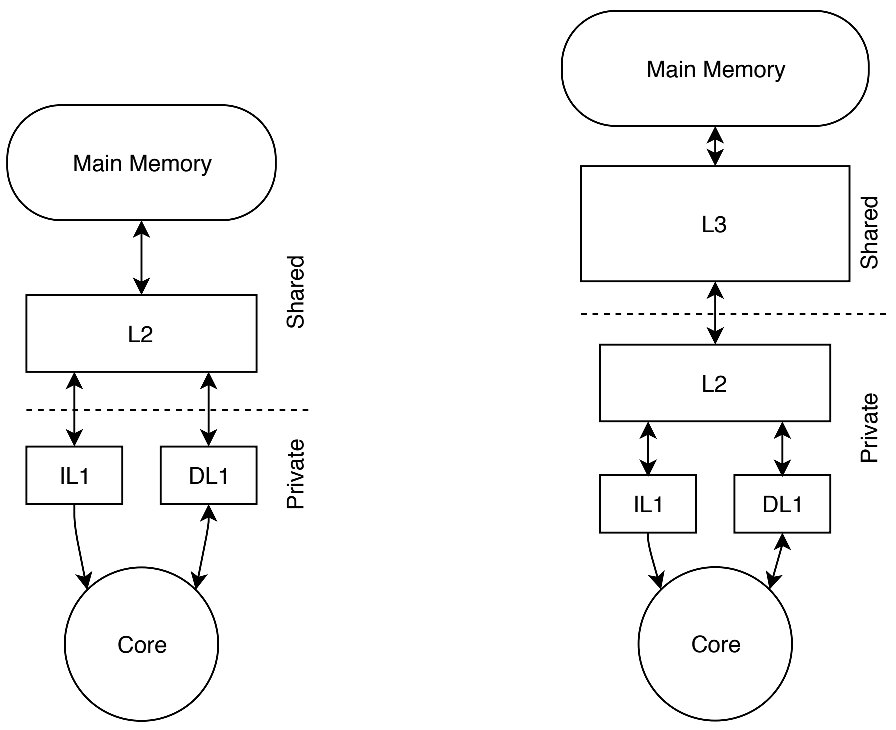
Figure 11. Execution times for 4096 × 4096 matrix transposition. Apart from the tile sizes in the x-axis, there can be seen the execution times for other T values, namely 6, 10, 12, 14, 20, 24, 28, 40, 48, 56, 96, 192, and 384.

6.4. Modelling and Reducing the Energy Consumption of the Memory Subsystem for the tiled Matrix Transposition

In this section we look at the energy consumption of the memory subsystems of the four platforms of Section 6.3. In order to do that, we use the analytical hit ratios introduced in Section 5 and combine them with the energy figures produced by the CACTI modeling tool 7.0 [20], which is able to compute energy per access and static power for a given cache or main memory configuration. Firstly, we outline the memory subsystems of the four platforms, introduce the energy and power figures of each component, and explain how to compute the overall energy. Secondly, we focus on a specific matrix size and test various tile sizes, obtaining the overall energy consumption and suggesting some hardware/software improvements.

6.4.1. Analytical Modeling of Energy Consumption

Figure 12 shows the two hierarchies found in the four platforms. Xeon-L5410 and 2QuadQ9550 have two levels of cache, and i7-2640M and i7-4810MQ have three. The last-level cache, closest to the main memory is shared among cores (only one is shown in Figure 12), while the lowest cache level(s) are private to cores. All data caches are copy-back caches, meaning that the effect of individual stores is not seen by upper levels until a whole dirty cache line is evicted. The load and store operations executed by the core access first to the L1 cache. On a miss, the next cache level is looked up until a hit is found or main memory is reached. A last-level cache miss brings the missed line at least to the L1 cache and depending on the inter-level content management policy also to L2 and L3 caches. For the sake of simplicity, though the four platforms differ in such a policy, we assume for all platforms an inclusive policy, which dictates the replication of lines brought from main memory to all cache levels, that is, L1 $\subset$ L2 $\subset$ L3.



(**a**) 2 Levels: QuadQ9550 and XeonL5410 (**b**) 3 Levels: i7-2640 and i7-4810

Figure 12. Memory hierarchy organizations.

Table 2 shows the cache and technological parameters for all platforms. Energy and power figures have been computed by CACTI, considering the technological node, the memory level, and the different sizes and associativities. The energy to read/write an 8B element is reported only in L1 caches, where load and store instructions operate at this granularity, whereas the energy to read/write 64B objects allows accounting for the cache line transfers among all the memory levels. We chose a 1 GiB main memory because it is enough to store 128 MiB, the size of the 4096×4096 matrix with 8-byte elements used in Figure 11 and in the next section.

Now we can use the hit ratio models in Section 5 to count all events of interest at any level of the inclusive hierarchy outlined above: #ld, #st, #hits, #replacements, and so forth. Then we can compute overall dynamic energy by multiplying event count (#ld, #st, #replacements, etc.) by the corresponding event energy as listed in Table 2. In order to compute the static energy we integrate all the leakage power during the execution time measured on each platform for each tile size (Figure 11).

6.4.2. Energy Consumption Estimation and Possible Improvements

Figure 13 shows the energy consumption estimates for the four platforms, varying the tile size in multiples of the line size to always achieve the ideal hit ratio. The first observation is that older technologies (45 nm) result in a much higher energy consumption, mostly due to the great amount of static consumption. Thus, in 45 nm, around 80% of the energy consumption is static, while in 32 and 22 nm this percentage gets reduced to around 50%. For all execution platforms, energy consumption is strongly correlated with the execution time of the matrix transposition algorithm. Thus, while we are at the optimal tile size, consumption remains constant for all platforms except for Q9550, where the increase in execution time triggers the static energy consumption as tile size increases.

Table 2. Memory subsystem parameters of the four platforms.

	2QuadQ9550	**i7-2640M**	**i7-4810MQ**	**XeonL5410**
Number of Cache Levels	2	3	3	2
Line Size	64 B	64 B	64 B	64 B
Technological node	45 nm	32 nm	22 nm	45 nm
L1 Size	32 KiB	32 KiB	32 KiB	32 KiB
L1 Associativity	8 way	8 way	8 way	8 way
L1 Read energy (pJ) 8 B	19.3	13.3	6.9	19.3
L1 Read energy (pJ) 64 B	665.3	464.8	327.5	66.5
L1 Write energy (pJ) 8 B	24.2	16.3	8.2	24.2
L1 Write energy (pJ) 64 B	670.3	468.2	333.0	67.0
L1 Leakage Power (mW)	65.2	45.2	20.4	65.2
L2 Size	6 MiB	256 KiB	256 KiB	12 MiB
L2 Associativity	24 way	8 way	8 way	24 way
L2 Read energy (nJ) 64 B	3.7	0.8	0.4	23.0
L2 Write energy (nJ) 64 B	4.0	0.8	0.5	25.1
L2 Leakage Power (W)	10.6	0.3	0.1	16.2
L2 Inclusion Policy simulated (actual)	Inclusive	Inclusive (non-inclusive)	Inclusive (non-inclusive)	Inclusive
L3 Size	N/A	4 MiB	6 MiB	N/A
L3 Associativity	N/A	16 way	32 way	N/A
L3 Read energy (nJ) 64 B	N/A	2.3	1.5	N/A
L3 Write energy (nJ) 64 B	N/A	2.4	1.6	N/A
L3 Leakage Power (W)	N/A	2.4	1.7	N/A
L3 Inclusion Policy	N/A	Inclusive	Inclusive	N/A
Main Memory Size (DDR3)	1 GiB	1 GiB	1 GiB	1 GiB
MM Read energy (nJ) 64 B	17.7	17.7	17.7	17.7
MM Write energy (nJ) 64 B	17.7	17.7	17.7	17.7
MM Leakage Power (W)	2.2	2.2	2.2	2.2

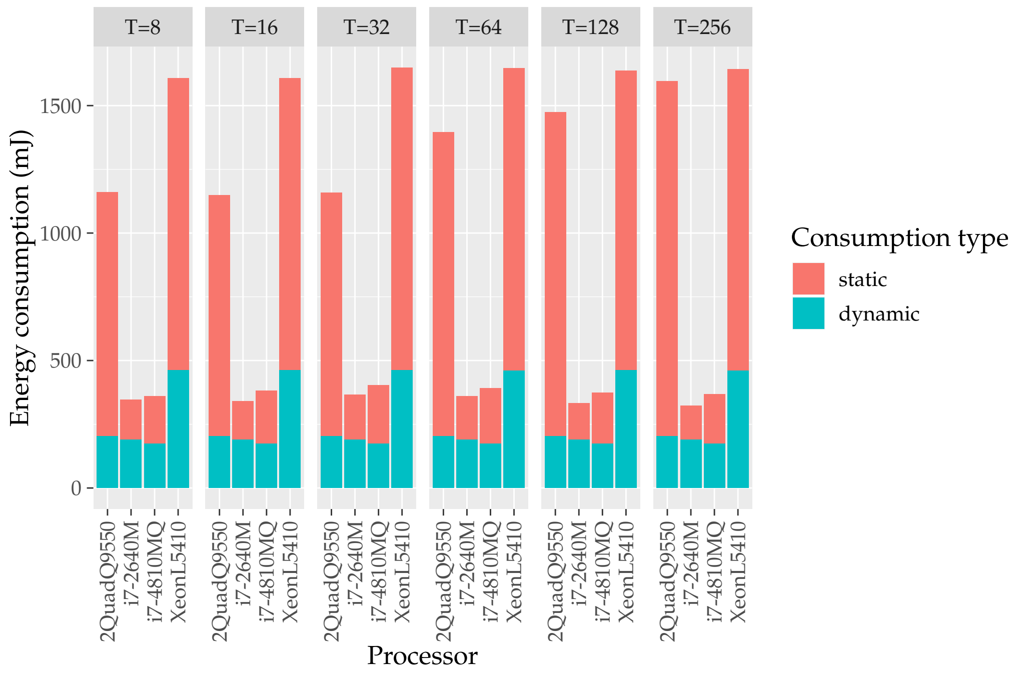


Figure 13. Energy consumed by the memory hierarchy of the experimental systems for different tile sizes.

Matrix transposition is a linear algebra operation that is usually used as part of a sequence to solve a major problem, such as in Reference [1,2]. In a supercomputing scenario, the size of the input data is big enough to fit into none of the levels of the cache hierarchy. In our case, a modest array of 4096 $\times$ 4096 elements (8 bytes each) occupies 128 MiB. In addition, the tiled transposition algorithm ensures that each visited element is never visited again. This can also occur in other linear algebra tiled algorithms as a vector-matrix multiplication. Although in the tiled matrix transposition the elements are only used effectively from L1, the inclusion and replacement policies keep copies of the elements already used in the different cache levels. Thanks to our analytical energy model, we can foresee how much energy could be saved if we were able to bypass the intermediate levels of the cache hierarchy when we know that their work is unneeded. So, we propose two possible energy-saving bypassing approaches. As a first approach, ensure that L1 cache misses are only brought to L1, and L1 replacements are evicted directly to main memory. In this approach, we save the activity associated with the other cache levels but not their static energy consumption, because we want them to continue servicing other applications that may be running in the system. As a second approach, besides the bypass strategy, we propose putting all cache levels but L1 in a drowsy state. In this state, the power supply voltage is reduced, which significantly reduces static consumption, but without losing the stored content as would occur when powering off. The disadvantage of this drowsy state is that it is not possible to resume the normal cache operation before reestablishing the nominal voltage. For instance, at 32 nm we would save up to 56% of static energy consumption [21,22]. Table 3 shows the percentage of energy savings in the memory hierarchy for each bypassing approach, relative to the results of Figure 13. The savings vary widely among the platforms and range from 3% to 46%. As expected, the second approach wins on all platforms, especially on the Q9550 and Xeon L5410, where static consumption dominates. On the other hand, i7-2640 and i7-4810 platforms show more contained consumption reductions. This is because their cache hierarchies are not as large as the Xeon L5410 (12 MiB), and also due to the technological improvement.

Table 3. Percentage of energy consumption savings on the memory hierarchy for our two bypassing scenarios.

	2QuadQ9550						i7-2640M					
	8	16	32	64	128	256	8	16	32	64	128	256
Scenario 1	4.2	4.3	4.2	3.5	3.3	3.1	10.7	10.9	10.2	10.3	11.1	11.5
Scenario 2	34.2	34.1	34.2	34.5	34.6	34.8	21.7	21.6	21.8	21.8	21.6	21.5
	i7-4810MQ						XeonL5410					
	8	16	32	64	128	256	8	16	32	64	128	256
Scenario 1	6.5	6.1	5.8	6.0	6.3	6.4	19.1	19.1	18.6	18.6	18.8	18.7
Scenario 2	16.8	17.0	17.1	17.1	16.9	16.9	46.6	46.6	46.5	46.5	46.5	46.5

In a supercomputing scenario, where the processor is running the matrix transposition algorithm intensively/exclusively, previous reductions in energy consumption will be multiplied by the thousands of computing nodes that may be running this algorithm at the same time. Due to the size of the data handled by the matrix transposition algorithm and the fact that the entire reuse of this algorithm is exploited from L1, the intermediate levels of cache do not help on improving performance. Moreover, they cause a noticeable energy-consumption overhead. The industry seems to have noticed that there are algorithms that do not show a conventional reuse and cause overheads in the memory subsystem. For this reason, since instruction extension SSE 4.1, Intel offers hint instructions (Non-Temporal Aligned Hint) for accessing to memory without polluting the cache hierarchy. Unfortunately, these instructions do not even store the value in L1, which in the case of tiled algorithms can make them unusable for this purpose. Another example is the upcoming CLDEMOTE instruction, which will be supported in future "Tremont and later" microarchitectures.

The CLDEMOTE instruction is a hint to the hardware that might help to move a cache line from the cache level(s) closest to the core to a cache level that is further from the core. However, what would be required to fit matrix transposition would be a CLDEMOTE variant to send directly to main memory the target line on L1 eviction.

7. Conclusions

In this paper, we study the data hit ratio for the matrix transposition problem. We first obtain an upper bound of the ideal data hit ratio independently of the transposition algorithm and cache configuration by calculating the minimum number of compulsory misses. Then, we analyze theoretically the relation between the LRU cache parameters and tile size to reach the ideal data hit ratio considering several padding options.

Our theoretical results show that, with a correctly configured tile (tile dimension equal to the cache line size), the ideal data hit ratio is reached by a set-associative cache with very few sets and no more than two ways. Such results are confirmed by means of extensive simulations, also validating the additional cache requirements when the tile is not correctly set. Also, we compare our results to that of a cache oblivious algorithm, which presents a very good performance independently of the cache configuration. Our results show that both the tiling (correctly configured) and the cache oblivious algorithms require two ways to reach the ideal data hit ratio for a cache with enough sets. If the number of sets in the data cache is too small, the tiling solution outperforms the cache oblivious one, which requires a cache with more ways to reach the ideal data hit ratio. So, tiling provides an equal or better hit ratio and at the same time avoids the recursive nature (in general less efficient) of cache oblivious algorithms.

Finally, in order to test the general performance and not only the hit ratio, we study the execution time of the matrix transposition on different machines. Although many current CPUs use pseudo-LRU instead of the LRU policy, our results show that their data hit ratio is almost identical. So, variations in their execution time reflect aspects such as data prefetchers and number of executed instructions. Also, we model the memory hierarchies of these machines to evaluate their energy consumption during the execution of matrix transposition. Our evaluation shows that conventional memory hierarchies offer no specific support to a well-tuned transposition, and we propose two bypassing approaches that reduce the energy consumption without any negative impact. Indeed, bypassing is likely to improve the performance of other running tasks in the system, since it avoids the unnecessary pollution that matrix transposition would impose otherwise.

The analytical expressions for the optimal configuration of a data cache for matrix transposition obtained in our research can be directly applied to such divergent scenarios as supercomputing and hard real-time systems. In the first case, the appropriate parameters in the algorithm will ensure an optimal use of the supercomputing system in terms of performance and efficiency of the resources used. In the second case, an accurate calculation of the number of hits and misses depending on cache parameters and tile size are essential for the worst-case execution time (WCET) computation in real-time systems.

Author Contributions: Conceptualization, C.R.; software, A.P.-Z. and C.R.; validation, A.P.-Z., C.R. and J.S.; investigation, A.P.-Z. and R.G.T.; writing–original draft preparation, A.P.-Z., J.S. and R.G.T.; writing–review and editing, A.P.-Z., J.S., R.G.T., V.V.-Y. and C.R.; visualization, J.S., A.P.-Z. and R.G.T.; supervision, J.S., C.R. and V.V.-Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by grants FPU14/02463, TIN2016-76635-C2-1-R (AEI/ERDF, EU), Aragón Government (T58_17R research group), and ERDF 2014-2020 «Construyendo Europa desde Aragón».

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Rearden, B.T.; Jessee, M.A. *SCALE Code System*; Technical Report; Oak Ridge National Laboratory (ORNL): Oak Ridge, TN, USA, 2016.
2. Ballard, S.; Hipp, J.; Kraus, B.; Encarnacao, A.; Young, C. GeoTess: A Generalized Earth Model Software Utility. *Seismol. Res. Lett.* **2016**, *87*, 719–725. [CrossRef]
3. Chatterjee, S.; Sen, S. Cache-efficient matrix transposition. In Proceedings of the Sixth International Symposium on High-Performance Computer Architecture HPCA-6 (Cat. No.PR00550), Toulouse, France, 8–12 January 2000; pp. 195–205. [CrossRef]
4. Lam, M.D.; Rothberg, E.E.; Wolf, M.E. The Cache Performance and Optimizations of Blocked Algorithms. *SIGPLAN Not.* **1991**, *26*, 63–74. [CrossRef]
5. Bao, W.; Krishnamoorthy, S.; Pouchet, L.N.; Sadayappan, P. Analytical Modeling of Cache Behavior for Affine Programs. *Proc. ACM Program. Lang.* **2017**, *2*, 32:1–32:26. [CrossRef]
6. Reineke, J.; Grund, D.; Berg, C.; Wilhelm, R. Timing Predictability of Cache Replacement Policies. *Real-Time Syst.* **2007**, *37*, 99–122. [CrossRef]
7. Zhong, Y.; Dropsho, S.G.; Shen, X.; Studer, A.; Ding, C. Miss Rate Prediction Across Program Inputs and Cache Configurations. *IEEE Trans. Comput.* **2007**, *56*, 328–343. [CrossRef]
8. Tsifakis, D.; Rendell, A.P.; Strazdins, P.E. *Cache Oblivious Matrix Transposition: Simulation and Experiment. Computational Science - ICCS 2004*; Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 17–25.
9. Frigo, M.; Leiserson, C.E.; Prokop, H.; Ramachandran, S. Cache-Oblivious Algorithms. In Proceedings of the 40th Annual Symposium on Foundations of Computer Science FOCS '99, New York, NY USA, 17–18 October 1999; IEEE Computer Society: Washington, DC, USA, 1999; p. 285.
10. Yotov, K.; Roeder, T.; Pingali, K.; Gunnels, J.; Gustavson, F. An Experimental Comparison of Cache-oblivious and Cache-conscious Programs. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA '07, New York, NY, USA, 9–11 June 2007; ACM: New York, NY, USA, 2007; pp. 93–104. [CrossRef]
11. Leiserson, C.E. Cache-Oblivious Algorithms. In *Algorithms and Complexity*; Petreschi, R., Persiano, G., Silvestri, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; p. 5.
12. Frigo, M.; Leiserson, C.E.; Prokop, H.; Ramachandran, S. Cache-Oblivious Algorithms. *ACM Trans. Algorithms* **2012**, *8*. [CrossRef]
13. Hong, C.; Bao, W.; Cohen, A.; Krishnamoorthy, S.; Pouchet, L.N.; Rastello, F.; Ramanujam, J.; Sadayappan, P. Effective Padding of Multidimensional Arrays to Avoid Cache Conflict Misses. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '16, Santa Barbara, CA, USA, 13–17 June 2016; ACM: New York, NY, USA, 2016; pp. 129–144. [CrossRef]
14. Panda, P.R.; Nakamura, H.; Dutt, N.D.; Nicolau, A. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Trans. Comput.* **1999**, *48*, 142–149. [CrossRef]
15. Bacon, D.F.; Chow, J.H.; Ju, D.C.R.; Muthukumar, K.; Sarkar, V. A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness. In Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research CASCON '94, Toronto, ON, Canada, 31 October–3 November 1994; IBM Press: Indianapolis, IN, USA, 1994; p. 3.
16. Morton, G.M. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*; Technical Report; IBM Ltd.: Ottawa, ON, Canada, 1966.
17. Abel, A.; Reineke, J. Measurement-based Modeling of the Cache Replacement Policy. In Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Philadelphia, PA, USA, 9–11 April 2013.
18. Berg, C. PLRU Cache Domino Effects. In Proceedings of the Workshop On Worst-Case Execution Time (WCET) Analysis, Dresden, Germany, 4 July 2006.
19. Abel, A.; Reineke, J. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2014), Monterey, CA, USA, 23–25 March 2014; IEEE Computer Society: Washington, DC, USA, 2014; pp. 141–142. [CrossRef]

20. Balasubramonian, R.; Kahng, A.B.; Muralimanohar, N.; Shafiee, A.; Srinivas, V. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* **2017**, *14*, 14:1–14:25. [CrossRef]
21. Fitzgerald, B.; Lopez, S.; Sahuquillo, J. Drowsy cache partitioning for reduced static and dynamic energy in the cache hierarchy. In Proceedings of the 2013 International Green Computing Conference Proceedings, Arlington, VA, USA, 27–29 June 2013; pp. 1–6. [CrossRef]
22. Flautner, K.; Kim, N.S.; Martin, S.; Blaauw, D.; Mudge, T. Drowsy caches: Simple techniques for reducing leakage power. In Proceedings of the 29th Annual International Symposium on Computer Architecture, Anchorage, AK, USA, 25–29 May 2002; pp. 148–157. [CrossRef]