Cache Oblivious Matrix Transposition: Simulation and Experiment

Dimitrios Tsifakis, Alistair P. Rendell* and Peter E. Strazdins
Department of Computer Science, Australian National University
Canberra ACT0200, Australia

dtsifakis@ieee.org, alistair.rendell@anu.edu.au
peter.strazdins@anu.edu.au

Abstract. A cache oblivious matrix transposition algorithm is implemented and analyzed using simulation and hardware performance counters. Contrary to its name, the cache oblivious matrix transposition algorithm is found to exhibit a complex cache behavior with a cache miss ratio that is strongly dependent on the associativity of the cache. In some circumstances the cache behavior is found to be worst than that of a naïve transposition algorithm. While the total size is an important factor in determining cache usage efficiency, the sub-block size, associativity, and cache line replacement policy are also shown to be very important.

1. Introduction

The concept of a cache oblivious algorithm (COA) was first introduced by Prokop in 1999 [1] and subsequently refined by Frigo and coworkers [2, 3]. The idea is to design an algorithm that has asymptotically optimal cache performance without building into it any explicit knowledge of the cache structure (or memory architecture) of the machine on which it is running. The basic philosophy in developing a COA is to use a recursive approach that repeatedly divides the data set until it eventually become cache resident, and therefore cache optimal. COA for matrix multiplication, matrix transposition, fast Fourier transform, funnelsort and distribution sort have been outlined (see [4] and references therein).

Although a number of COA have been proposed, to date most of the analyses have been theoretical with few studies on actual machines. An exception to this is a paper by Chatterjee and Sen (C&S) [5] on "Cache-Efficient Matrix Transposition". In this paper C&S outline a number of matrix transposition algorithms and compare their performance using both machine simulation and elapsed times recorded on a Sun UltraSPARC II based system. Their work is of interest in two respects; first their simulations showed that while the cache oblivious transposition algorithm had the smallest number of cache misses for small matrix dimensions, for large dimensions it was actually the worst. Second, their timing runs showed that in most cases the COA was significantly slower than the other transposition algorithms. It was suggested that the poor performance of the cache oblivious matrix transposition algorithm was related to the associativity of the cache, although this relationship was not fully explored.

Today virtually all modern processors include a number of special registers that can be programmed to count specific events. These so called "hardware performance counters", coupled with the availability of a number of portable libraries to access them [6, 7] means that it is now possible to gather very detailed information about how a CPU is performing. Examples of the sort of events that can be counted include machine cycles, floating point operations, pipeline stalls, cache misses etc. Using

these registers it is therefore possible to directly assess the performance of COA on real machines, and perform details studies comparing theoretical and observed performance. In this respect there have, very recently, appeared a number of studies looking at COA using hardware performance counters, e.g., cache oblivious priority queues [8, 9] and cache oblivious sorting [10, 11].

The primary aim of this paper is to explore further the cache oblivious matrix transposition algorithm with the aim of rationalizing the results of C&S [5]. To achieve this, a combination of machine simulation and hardware performance counters is used, and in this respect the work presented here compliments the other recent studies of COA [8–11].

2. Matrix Transposition

Matrix A of size $m \times n$ is transposed into a matrix B of size $m \times n$ such that:

$$A_{ij} = B_{ji} \quad \forall i \in [1 \cdots m],\ j \in [1 \cdots n]$$

Frequently the transposition occurs "in-situ", in which case the memory used for storing matrix A and B is identical. For the purpose of this paper the discussion will be restricted to square ($m=n$) in-situ matrix transpositions. Three different algorithms will be consider; cache ignorant, blocked, and cache oblivious.

2.1 Cache Ignorant Matrix Transposition

A naïve implementation of matrix transposition is given by the following C code:

```
for (i = 1; i < n; i++)
      for (j = 0; j < i; j++){
            tmp = A[j][i];
            A[i][j]=A[j][i];
            A[j][i]=tmp;      }
```

In this implementation the statements in the inner loop are executed $n(n-1)/2$ times and no special care is made to use the cache efficiently.

2.2 Cache Blocked Matrix Transposition

In the cache blocked transposition algorithm the matrix is effectively divided into a checkerboard of small blocks. Two blocks that are symmetrically distributed with respect to the leading diagonal are identified and their data is copied into cache resident buffers. The buffers are then copied back into the matrix, but in transposed form. Pseudo code illustrating this algorithm is given below:

```
for (i = 0; i < n; i += size)
   for (j = 0; j < i; j += size){
      copy A[i:i+size-1][j:j+size-1] to buf1
      copy A[j:j+size-1][i:i+size-1] to buf2
      transpose buf1 to A[j:j+size-1][i:i+size-1]
      transpose buf2 to A[i:i+size-1][j:j+size-1]}
```

In the above the dimension of the small blocks is given by `size` with the restriction that $2 \times \texttt{size}^2$ is less than the cache size, and it has been assumed that `size` perfectly divides the matrix dimension `n`. In contrast to the cache ignorant scheme, each

element of the matrix is now loaded into registers twice; once when copying the data from matrix `A` to `buf`, and once when copying each element from `buf` back to `A`.

2.3 Cache Oblivious Matrix Transposition

In the cache oblivious transposition the largest dimension of the matrix is identified and split, creating two sub-matrices. Thus if $n \geq m$ the matrices are partitioned as:

$$A = (A_1 A_2), \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

This process continues recursively until individual elements of A and B are obtained at which point they are swapped.

3. Performance Simulation

To analyse performance a basic cache simulator was written. This assumes a single level of cache, and includes parameters for the cache line size, the number of cache lines, the associativity, and the cache line replacement policy. Code to perform the different matrix transposition algorithms was written and annotated such that the memory address corresponding to every matrix element access was passed to the cache simulator, which then determined whether it was either a cache hit or miss.

When simulating the cache, a number of other issues also need to be considered; notably the initial alignment of the matrix with respect to the cache, the word size of each matrix element, and the dimension of the matrix. For simplicity in the following experiments the first element of the matrix is always aligned perfectly with the start of a cache line, the cache line size is a perfect multiple of the matrix element word size, and the matrix dimensions are chosen such that different rows of the matrix never share the same cache line.

Before considering the results of the simulator experiments, it is useful to illustrate the typical access patterns of the three matrix transposition algorithms. This is shown in fig. 1. Of particular interest is the COA. This clearly shows a natural partitioning of the matrix into a hierarchy of square blocks of dimensions 2^x. Thus if the cache line size was sufficient to hold exactly 4 matrix elements and the total cache size was sufficient to hold 8 cache lines, then both of the shaded blocks shown in fig. 1.c could, in principle, reside in cache simultaneously and the algorithm would therefore be expected to show minimal cache misses.

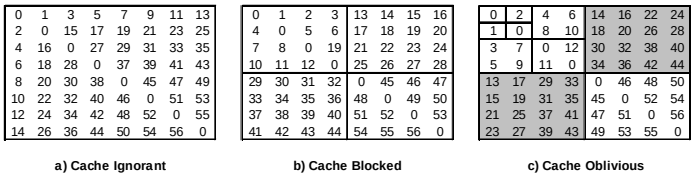

Figure 1: Typical access patterns for the three transposition algorithms on an 8×8 matrix (A blocking size of 4 is used in the cache blocked algorithm).

In their paper C&S [5] presented a table of cache misses for a variety of different matrix transpositions algorithms and for four different matrix sizes. Their simulated

results for the cache ignorant, cache blocked (full copy), and COA are reproduced in table 1. The strange behavior of the COA is immediately obvious; for N=1024 it has the lowest number of cache misses, while for N=8192 it has the largest.

	Matrix Dimension			
Algorithm	**1024**	**2048**	**4096**	**8192**
Cache ignorant	589795	2362002	9453724	37826712
Full copy cache blocked	275550	1170003	4804808	19493808
Cache oblivious	131226	923295	7101600	56158873

Table 1: Cache misses for three matrix transposition algorithms. Data taken from C&S [5] and obtained by simulating a 16KB direct mapped cache with a 32byte cache line. The matrix is square with a 4byte word size.

In fig. 2, the simulations of C&S [5] have been extended to include all matrix dimensions that are less than 10,000 but that are multiples of the cache line size. The figure includes data for the cache ignorant and COA, and also the minimum and maximum number of cache misses. The minimum cache miss ratio assumes all data in a cache line is fully utilized before that cache line is evicted, while the maximum cache miss ratio assumes a cache miss occurs for every read, but the subsequent write is a cache hit. Assuming there are no cache line conflicts between the temporary buffers and the matrix elements then the cache blocked algorithm will essentially give the minimum number of cache misses.

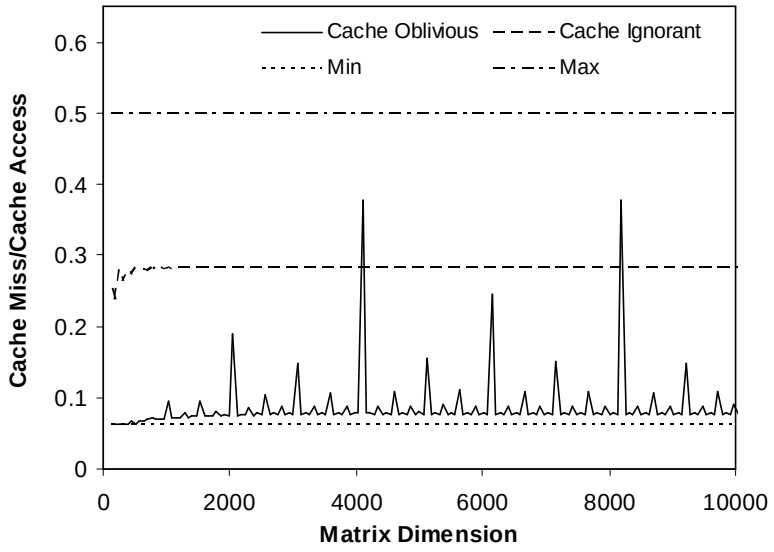

Figure 2: Simulated cache miss to access ratio for cache oblivious and cache ignorant matrix transposition algorithms, using a 16KB, direct mapped cache with a 32byte line size and 4byte matrix elements. Matrix dimensions are always an exact multiple of the cache line size.

From fig. 2, it is apparent that the COA is far from cache oblivious. Rather, the cache miss profile shows significant structure. Furthermore the data points chosen by C&S (N=1024, 2048, 4096 and 8192) [5] are actually some of the worst possible values; for many other dimensions the COA achieves close to the minimum.

The poor performance of the COA for N=4096 and 8192 is due to the fact that for both of these dimensions one row of the matrix is an exact multiple of the cache size. With a direct mapped cache this means that elements in the same column of the matrix map to the same cache line. Inspecting the access pattern for the COA given in fig. 1 clearly shows that this will be a problem. For example, if a cache line is assumed to hold 4 matrix elements and the matrix is aligned such that accesses {13, 17, 29, 33} correspond to one cache line, then to fully utilize the data in this cache line there must be no cache line conflicts between accesses 13 and 33. However, between these accesses 7 other cache lines will be accessed – corresponding to accesses {15,19,31,35}, {21,25,37,41} {23,27,39,43}, {14,16,22,24}, {18,20,26,28}, {30,32,38,40}, and {34,36,42,44}. The first three of these share the same cache line as the initial access, while the latter 4 will share another cache line. Changing the matrix row size to be, e.g., 1.5 times the cache size will halve the number of cache line conflicts, but will not totally eliminate then. Similar effects occur for other partial multiples giving the complicated structure shown in fig. 2.

From the above discussion increasing cache line associativity should lead to a decrease in the number of cache line conflicts. This is demonstrated by the simulated results in fig. 3. It is interesting to note, however, that the reduction in cache misses is not universal for all matrix dimensions. Thus while the cache miss ratio for N=4096 and 8192 decreases in going from a direct to 2-way set associative cache, the cache miss ratio for N=6144 actually increases slightly. This effect is due to the fact that increasing the cache line associativity while maintaining the same total cache size actually doubles the number of possible cache line conflicts, although also providing two possible cache lines that can be used to resolve each conflict. For example, whereas with the direct mapped cache, a cache line conflict was encountered every 4096 matrix elements and could not be avoided, with a 2-way set associative cache a conflict arises every 2048 elements but there are two possible cache line locations that can be used to remove those conflicts. Thus predicting the overall effect of increasing cache line associativity is hard, although it appears beneficial overall.

Interestingly with an 8-way set associative cache, the data points that originally gave rise to the worst cache miss ratio, i.e. N=4096 and 8192, now give rise to the minimum. This is evident in fig. 3 as slight dips in the cache miss ratios for these data points. The existence of “magic dimensions” is not surprising; with a 4-way associative cache the cache line conflict discussed above for accesses {13,17,29,33}, {15,19,31,35}, {21,25,37,41} and {23,27,39,43} would be removed. If these assesses also conflicted with those of {14,16,22,24}, {18,20,26,28}, {30,32,38,40}, and {34, 36,42,44}, an 8-way set associative cache would be required to remove the conflict.

This result can be generalized for a cache whose line size (l) is a power of 2. Assuming that each matrix row starts with a new line, a COA will attain minimum misses if its associativity is at least 2^l. This is because it will reach a stage where it will swap two $l \times l$ blocks, which will be stored in $2 \times l$ lines. Providing a least recently

used (LRU) replacement policy is used, the cache will be able to hold all of these simultaneously. If matrix rows are not aligned with cache lines, the two sub-blocks will be stored in at most $4 \times l$ lines; in this case, an associativity of $4 \times l$ would be required in order to minimize cache misses

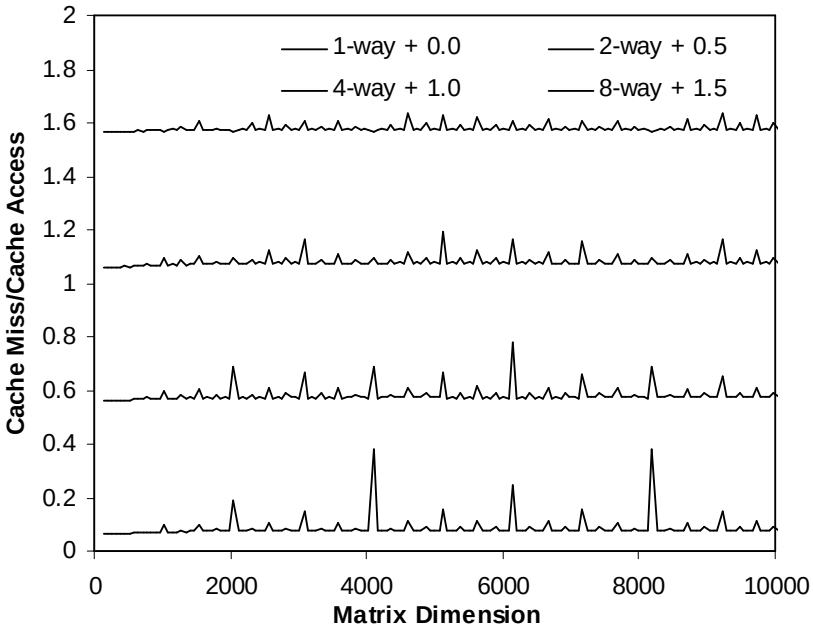

Figure 3: Simulated cache miss to access ratio as a function of cache line associtivity for the cache oblivious matrix transposition algorithms using a 16KB cache with a 32byte line size and 4byte matrix elements. Matrix dimensions are chosen to be a direct multiple of the cache line size.

4. Performance Measurements

Using hardware performance counters cache miss data was gathered for:

- A 167MHz Sun UltraSPARC I system with a 16KB direct mapped L1 data cache with 32-byte cache line and a 512KB level 2 cache
- A 750MHz Sun UltraSPARC III system with a 64KB 4-way set associative L1 data cache with a 32-byte cache line size and an 8MB level 2 cache

The Sun UltraSPARC I system has a direct mapped level 1 cache with identical structure to that used by C&S [5]. The measured and simulated cache misses for the COA are given in table 3. The matrix elements are 4 bytes, with data given for dimensions around N=4096 and 8192. Two different simulated results are shown; for Sim#1 the cache line size is 32bytes while for Sim#2 it is 16bytes. This is done since the 32byte Ultra SPARC I cache line is actually split into two 16byte sub-blocks, and halving the cache line size in the simulated results is an attempt to approximately (but not exactly) account for this effect.

	Cache Misses				Cache Misses		
N	Ultra I	Sim#1	Sim#2	N	Ultra I	Sim#1	Sim#2
4080	5270060	2391376	4379033	8176	21408816	9646976	17621793
4088	5199068	2316901	4331751	8184	20438710	9301155	17393199
4096	12997199	12615680	12595200	8192	52589636	50479104	50397184
4104	6857957	4176906	4849009	8200	27268615	16677803	19370810
4112	5499487	2550567	4543409	8208	21899111	10117444	18110294

Table 3: Measured and simulated cache misses on the Ultra SPARC I for square in-situ COA and a variety of matrix dimensions (N). Simulated results reported with both 32byte (Sim#1) and 16byte (Sim#2) cache line size

The results as measured by the hardware performance counters clearly show a large number of cache misses at N=4096 and 8192, that decreases markedly for matrix dimensions that are either slightly smaller or larger. At these dimensions both the experimental and simulated results are approximately identical – reflecting the fact that essentially every matrix access results in a cache miss. For other dimensions the simulated results obtained using a 16byte cache line are closest to the experimentally recorded results, with the experimental results showing slightly higher number of cache misses. This is to be expected since the simulated results with a 16kbyte cache and a 16byte cache line has twice the number of cache lines as a 16kbyte cache with a sub-blocked 32byte cache line and is therefore a more flexible cache model. It should also be noted that the results from the hardware counters show some sensitivity to the choice of compilation flags; the above results were obtained using the `-fast` option and if this is lowered to `-x01` the agreement between the measured and simulated number of cache misses actually improves slightly.

	Cache Misses				Cache Misses		
N	Ultra III	Sim#LRU	Sim#Ran	N	Ultra III	Sim#LRU	Sim#Ran
1000	263477	258290	265990	4072	4361124	4283628	4391355
1024	375028	262983	284788	4096	7751760	4232722	5973761
1048	289398	284628	292332	4120	4464945	4382333	4496696
2024	1075296	1058128	1083519	8168	17577433	17234628	17669072
2048	1923256	1056544	1491917	8192	30873556	16956716	23904911
2072	1128147	1108024	1136952	8216	17791597	17425642	17882604

Table 4: Cache misses on an UltraSPARC III system. Simulated results using LRU (Sim#LRU) and random (Sim#Ran) cache replacement policy for the square in-situ COA and a variety of different matrix dimensions (N).

In table 4, similar cache miss data is given for the Ultra SPARC III platform. On this system there is a 4-way set associative level 1 cache. From the results given in section 3, it might be expected that there would be little difference between the number of cache misses that occurs for N=4096 or 8192 and surrounding values of N. The experimental results show, however, that this not the case; rather, the number of cache misses is roughly double at these values of N compared to those at nearby values of N. This is due to the cache line replacement policy on the UltraSPARC III, which is pseudo random rather than LRU [12]. Simulated results using a random number generator to determine cache line placement are shown as "Sim#Ran" in table 4. These show a considerable increase in the number of cache misses when N=1024,

2048, 4096 and 8192, although still somewhat less than those recorded by the hardware performance counters. Outside these data points there appears to be little difference between the use of an LRU or random cache line replacement policy.

5. Conclusions

The performance of a COA for matrix transposition has be analyzed, with respect to cache misses, via both simulation and use of hardware performance counters on two fundamentally different UltraSPARC systems. The results confirm earlier work by C&S [5] showing very high numbers of cache misses for certain critical matrix dimensions. In general it was shown that the cache miss characteristics of the "cache oblivious" matrix transposition algorithm has significant structure, the form of which depends on a subtle interplay between cache size, matrix dimension, number of matrix elements per cache line, cache line size, cache associativity and the cache line replacement policy. Predicting, a priori, when the COA will perform well and when it will perform poorly is non-trivial, although increased cache line associativity appears overall to be beneficial.

The work presented here has only been concerned with the cache usage characteristics of cache oblivious matrix transposition. The observed performance of any algorithm is of course dependent on other factors as well as efficient cache usage. Details of this will be discussed in a subsequent publication.

Acknowledgements: APR and PES acknowledge support from Australian Research Council Linkage Grant LP0347178 and Sun Microsystems. Discussions with Bill Clarke and Andrew Over are also gratefully acknowledged.

References

1. H. Prokop, *Cache-Oblivious Algoirthms*, MSc Thesis, Dept. Electrical Eng. and Computer Science, Massachusetts Institute of Technology, 1999
2. M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, *Cache-Oblivious Algoirthms (extended abstract)*, Proceedings of the 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Science Press, 285-297, 1999.
3. M. Frigo, *Portable High Performance Programs*, PhD Thesis, Dept. Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1999.
4. E.D. Demaine, *"Cache-Oblivious Algoirthms and Data Structures"*, Lecture notes in Computer Science, BRICS, University of Aarhus, Denmark June 27-July 1, 2002.
5. S. Chatterjee and S. Sen, *Cache-Efficient Matrix Transposition*, Proceedings of the 6th International Conference on High Performance Computing Architecture, 195, 2000
6. Performance Application Programmer Interface (PAPI) http://icl.cs.utk.edu/ projects/papi
7. Performance Counter Library (PCL), http://www.fz-juelich.de/zam/PCL
8. J.H. Olsen and S.C. Skov, *Cache-Oblivious Algoritsms in Practice*, MSc Thesis, Dept Computing, University of Copenhagen, 2002
9. L. Arge, M. Bender, E. Demaine, B. Holland-Minkley and J. Munro, *Cache-Oblivious Priority Queue and Graph Algorithhm Applications*, Submitted to SIAM journal on Computing, May 2003.
10. F. Rønn, *Cache-0blivious Searching and Sorting*, MSc thesis, Dept Computer Science, University of Copenhagen, July 2003.
11. K. Vinther, *Engineering Cache-Oblivious Sorting Algoirthms*, MSc Thesis, Dept. Computer Science, University of Aarhus, June 2003.
12. D. May, R. Pas and E. Loh, *The RWTH SunFire SMP-Cluster User's Guide (version 3.1)*, http://www.rz.rwth-aachen.de/computing/info/sun/primer/ primer_V3.1.html, July 2003