Tuning Blocked Array Layouts to Exploit Memory Hierarchy in SMT Architectures

Evangelia Athanasaki, Kornilios Kourtis, Nikos Anastopoulos, and Nectarios Koziris

National Technical University of Athens,
School of Electrical and Computer Engineering,
Computing Systems Laboratory,
{valia, kkourt, anastop, nkoziris}@cslab.ece.ntua.gr

Abstract. Cache misses form a major bottleneck for memory-intensive applications, due to the significant latency of main memory accesses. Loop tiling, in conjunction with other program transformations, have been shown to be an effective approach to improving locality and cache exploitation, especially for dense matrix scientific computations. Beyond loop nest optimizations, data transformation techniques, and in particular blocked data layouts, have been used to boost the cache performance. The stability of performance improvements achieved are heavily dependent on the appropriate selection of tile sizes.

In this paper, we investigate the memory performance of blocked data layouts, and provide a theoretical analysis for the multiple levels of memory hierarchy, when they are organized in a set associative fashion. According to this analysis, the optimal tile size that maximizes L1 cache utilization, should completely fit in the L1 cache, even for loop bodies that access more than just one array. Increased self- or/and cross-interference misses can be tolerated through prefetching. Such larger tiles also reduce mispredicted branches and, as a result, the lost CPU cycles that arise. Results are validated through actual benchmarks on an SMT platform.

1 Introduction

The ever increasing gap between processor and memory speed, necessitates the efficient use of memory hierarchy to improve performance on modern microprocessors [15]. Compiler optimizations can efficiently keep reused data in memory hierarchy levels close to processors. Loop tiling is one of the well-known control transformation techniques, which, in combination with loop permutation, loop reversal and loop skewing attempt to modify the data access order to improve data locality. Combined loop and data transformations were proposed to avoid any negative effect to the number of cache hits for some referenced arrays, while increasing the locality of references for a group of arrays.

The automatic application of nonlinear layouts in real compilers is a really time consuming task. It does not suffice to identify the optimal layout either blocked or canonical one for each specific array. For blocked layouts, we also

P. Bozanis and E.N. Houstis (Eds.): PCI 2005, LNCS 3746, pp. 600–610, 2005.

need an automatic and quick way to generate the mapping from the multidimensional iteration indices to the correct location of the respective data element in the linear memory. Any method of fast indexing for non-linear layouts will allow compilers to introduce such layouts along with row or column-wise ones, therefore further reducing memory misses. In [1], in order to facilitate the automatic generation of tiled code that accesses blocked array layouts, we proposed a very quick and simple address calculation method of the array indices. Our method has proved to be very effective at reducing cache misses.

In this paper, we further extend our previous work by providing a tile selection formula that applies to blocked array layouts. All related work selects tiles smaller than half of the cache capacity (they usually refer to L1 cache or cache and TLB concurrently). However, blocked array layouts almost eliminate self-interference misses, while cross-interference can be easily obviated. Therefore, other factors, before negligible, now dominate cache and TLB behaviour, that is, code complexity, number of mispredicted branches and cache utilization. We have managed to reduce code complexity of accesses on data stored in a blocked-wise manner by the use of efficient indexing, described in detail in [1]. Experimentation has proved that maximum performance is achieved when L1 cache is fully utilized. At this point, tile sizes fill the whole L1 cache. Proper array alignment obviates cross-conflict misses, while the whole cache is exploited, as all cache lines contain useful data. Such large tiles reduce the number of mispredicted branches, as well. Experimental results were conducted using the Matrix Multiplication.

The remainder of the paper is organized as follows: Section 2 briefly reviews the related work. Section 3 presents a theoretical analysis of cache performance and demonstrates the need of optimizing L1 cache behaviour, as it is the dominant factor on performance. A tight lower bound for cache and TLB misses is calculated, which meets the access pattern of the Matrix Multiplication kernel. Section 4 illustrates execution results of optimized numerical codes, giving heuristics of tile size selection. Finally, concluding remarks are presented in Section 5.

2 Related Work

Loop tiling is a control transformation technique that partitions the iteration space of a loop nest into blocks in order to reduce the number of intervening iterations and thus data fetched between data reuses. This allows reused data to still be in the cache or register file, and hence reduces memory accesses. Without tiling, contention over the memory bus will limit performance. Conflict misses [20] may occur when too many data items map to the same set of cache locations, causing cache lines to be flushed from cache before they may be used, despite sufficient capacity in the overall cache. As a result, in addition to eliminating capacity misses [11], [23] and maximizing cache utilization, the tile should be selected in such a way that there are no (or few) self conflict misses, while cross conflict misses are minimized [3], [4], [5], [10], [17].

To model self conflict misses due to low associativity cache, [24] and [12] use the effective cache size $q \times C$ ($q < 1$), instead of the actual cache size C, while [3], [4], [10] and [19] explicitly find the non-conflicting tile sizes. Taking into account cache line size as well, column dimensions (without loss of generality, assume a column major data array layout) should be a multiple of the cache line size [4]. If fixed blocks are chosen, Lam et al. in [10] have found that the best square tile is not larger than $\sqrt{\frac{aC}{a+1}}$, where a = associativity. In practice, the optimal choice may occupy only a small fraction of the cache, typically less than 10%. What's more, the fraction of the cache used for optimal block size decreases as the cache size increases. The desired tile shape has been explicitly specified in algorithms such as [5], [3], [4], [23], [24], [10]. Both [23] and [10] search for square tiles. In contrast, [3], [4] and [24] find rectangular tiles or [5] even extremely tall tiles (the maximum number of complete columns that fit in the cache). However, extremely wide tiles may introduce TLB thrashing. On the other hand, extremely tall or square tiles may have low cache utilization.

Unfortunately, the performance of a tiled program resulting from existing tiling heuristics does not have robust performance [13], [17]. Instability comes from the so-called pathological array sizes, when array dimensions are near powers of two, since cache interference is a particular risk at that point. Array padding [8], [13], [16] is a compiler optimization that increases the array sizes and changes initial locations to avoid pathological cases. It introduces space overhead but effectively stabilizes program performance. Cache utilization for padded benchmark codes is much higher overall, since padding is used to avoid small tiles [17]. As a result, more recent research efforts have investigated the combination of both loop tiling and array padding in the hope that both magnitude and stability of performance improvements of tiled programs can be achieved at the same time. An alternative method for avoiding conflict misses is to copy tiles to a buffer and modify code to use data directly from the buffer [5], [10], [21]. Copying in [10] can take full advantage of the cache as it enables to use tiles of size $\sqrt{C} \times \sqrt{C}$ in each blocked loop nest. However performance overhead due to runtime copying is low if tiles only need to be copied once.

Cache behaviour is extremely difficult to analyze, reflecting its unstable nature, in which small modifications can lead to disproportionate changes in cache miss ration [20]. Traditionally, cache performance evaluation has mostly used simulation. Although the results are accurate, the time needed to obtain them is typically many times greater than the total execution time of the program being simulated. To try to overcome such problems, analytical models of cache behaviour combined with heuristics have also been developed, to guide optimizing compilers [6], [16] and [23], or study the cache performance of particular types of algorithm, especially blocked ones [3], [7], [10], and [22]. Code optimizations, such as tile size selection, selected with the help of predicted miss ratios require a really accurate assessment of program's code behaviour. For this reason, a combination of cache miss analysis, simulation and experimentation is the best solution for optimal selection of critical transformations.

The previous approaches assumed linear array layouts. However, as aforementioned studies have shown, such linear array memory layouts produce unfavorable memory access patterns, that cause interference misses and increase memory system overhead. In order to quantify the benefits of adopting nonlinear layouts to reduce cache misses, there exist several different approaches. In [18], Rivera et al. considers all levels of memory hierarchy to reduce L2 cache misses as well, rather than reducing only L1 ones. He presents even fewer overall misses, however performance improvements are rarely significant. Park et al. in [14] analyze the TLB and cache performance for standard matrix access patterns, when tiling is used together with block data layouts. Such layouts with block size equal to the page size, seem to minimize the number of TLB misses.

3 Theoretical Analysis

In this section we study the cache and TLB behaviour, while executing the matrix multiplication benchmark, which is the building block of many scientific applications. The analysis is devoted to set associative caches. Arrays are considered to be stored in memory according to the proposed blocked layouts, that is, elements accessed in consecutive iterations are found in nearby memory locations. Blocked layouts eliminate all self-conflict misses. We examine only square tiles. Such tile shapes are required for symmetry reasons, to enable the simplification of the benchmark code. As a result, while optimizing nested loop codes and selecting tile sizes, we should focus on dimishing the remaining factors that affect performance. The following analysis is an effort to identify such factors.

3.1 Machine and Benchmark Specifications

The optimized ([9], [16], [1]) matrix multiplication code has the following form:

```
for (ii=0; ii < N; ii+=T)
  for (jj=0; jj < N; jj+=T)
      for (kk=0; kk < N; kk+=T)
          for (i = ii; (i < ii+T && i < N); i++)
              for (j = jj; (j < jj+T && j < N); j++)
                  for (k = kk; (k < kk+T && k < N); k++)
                      C[i,k]+=A[i,j]*B[j,k];
```

Table 1 contains the symbols used in this section to represent the machine characteristics.

3.2 Data L1 Misses

In case of set associative caches, apart from capacity misses, neither self- nor cross-interference misses arise. Even 2-way set associativity is enough for kernel codes such as matrix multiplication, where data elements from three different arrays are retrieved. Array alignment should be carefully chosen, so that no more

Table 1. Table of hardware specifications and respective symbols used in this section

	Xeon DP	**symbol**
CPU freq. :	2,8GHz	
L1 cache :	16KB 8-way set asssoc.	C_{L1}
L1 line :	64B	L_1
L1 miss penalty:	4 clock cycles	c_{L1}
total L1 misses :		M_1
L2 cache :	1MB 8-way set asssoc.	C_{L2}
L2 line :	64B	L_2
L2 miss penalty:	18 clock cycles	c_{L3}
total L1 misses :		M_2
TLB entries (L1):	64 addresses	E (# entries)
page size :	4KB	P
TLB miss penalty:	30 clock cycles	c_{TLB}
total L1 misses :		M_{TLB}
mispred. branch penalty:	20 clock cycles	c_{br}

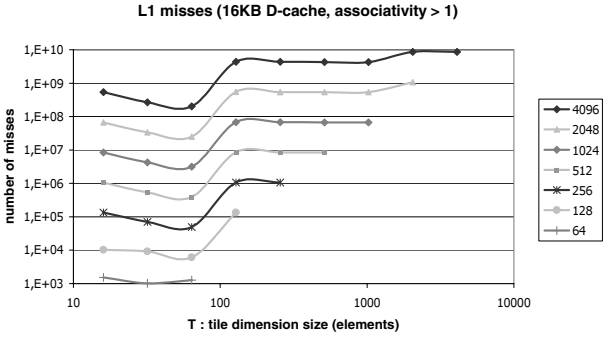

Fig. 1. Number of L1 cache misses on the Xeon DP architecture

than two arrays are mapped in the same cache location, concurrently. For this purpose, the starting mapping distances of arrays (that is, elements $A[0]$, $B[0]$, $C[0]$), should be chosen to be from L_1 to T^2 elements. Further analysis is beyond the scope of this paper.

$\boldsymbol{N^2 < C_{L1}}$**:** In this case, all three arrays can exploit reuse, both in-tile and intra-tile. For example, array C reuses one tile row along loop j (in-tile reuse) and a whole row of tile along loop jj (intra-tile reuse). In both cases, the working set fit in L1 cache. As a relult, for array C:

$M_C = x^2 \cdot \frac{T^2}{L_1} = \left(\frac{N}{T}\right)^2 \frac{T^2}{L_1} = \frac{N^2}{L_1}$

Similarly, $M_A = M_B = \frac{N^2}{L_1}$

$N^2 \geq C_{L1}$, $T \cdot N < C_{L1}$: As in the previous case: $M_A = M_C = \frac{N^2}{L_1}$

On the other hand, for array B only in-tile reuse can be exploited, as loop ii reuses N^2 elements, and the cache capacity is not adequate to hold them. As a result, each ii iteration will have to reload the whole array in the cache:

$M_B = x^3 \cdot \frac{T^2}{L_1} = \left(\frac{N}{T}\right)^3 \frac{T^2}{L_1} = \frac{N^3}{TL_1}$

In case that $N^2 = C_{L1}$, $T = N$, there is in fact no tiling, so reuse takes place in loops k and j for arrays A and C, containing just 1 element and one row of the array (N elements) respectively. As a result, reuse is exploited as above. However, the reference to array B reuses N^2 elements (the whole array) along loop i. In each iteration of i, two rows of B elements ($2N$ elements) have been discarded from the cache, due to references to arrays A and C. That is:

$M_B = \frac{N^2}{L_1} + (N-1) \cdot \frac{2N}{L_1}$

$N^2 > C_{L1}$, $3T^2 < C_{L1} \leq T \cdot N$: Three whole tiles fit in the cache, one for each of the three arrays. For arrays B, C reuse along loops ii and jj respectively can not be exploited, as there is not enough L1 cache capacity to hold N^2 and $T \cdot N$ elements respectively. The number of misses are:

$M_B = M_C = \frac{N^3}{TL_1}$

On the other hand, reuse along loop kk for array A can be exploited (only T^2 elements are included):

$M_A = \frac{N^2}{L_1}$

$N^2 > C_{L1}$, $T^2 \leq C_{L1} < 3T^2$: There is enough space for at most two whole tiles. Only in-tile reuse can be exploited in the arrays along the three inner loops. Thus:

$M_A = M_B = M_C = \frac{N^3}{TL_1}$

$N^2 > C_{L1}$, $T^2 > C_{L1} > T$: As in the previous case, no whole-tile reuse can be exploited. Additionally, in array B, in-tile reuse (along loop i) can not be exploited, either. Therefore, the total number of misses for each array is:

$M_A = M_C = \frac{N^3}{TL_1}$

$M_B = \frac{N^3}{L_1}$

Summary of the Data L1 Misses: Figure 1 illustrates the graphic representation of the total number of Data L1 cache misses ($M_1 = M_A + M_B + M_C$) for different problem sizes. The cache capacity and organization have the characteristics of the Xeon DP architecture (table 1).

L1 cache misses increase sharply when the working set, reused along the three innermost loops, overwhelms the L1 cache. That is, the tile size overexceeds the L1 cache capacity (C_{L1}), and no reuse can be exploited for at least one array.

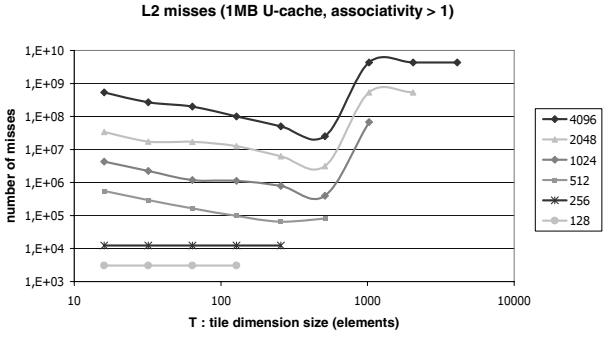

Fig. 2. Number of L2 cache misses on the Xeon DP architecture

3.3 L2 Misses

This cache level has similar behaviour as the L1 cache. As a result, we skip the detailed analysis and provide only with the corresponding graphs. Figure 2 presents the number of L2 cache misses in case of a set associative cache, with size equal the L2 cache of the Intel Xeon platform (table 1).

We note that L2 cache is unified (for data and instructions). However, the number of misses hardly increases (less than 1%) compared to an equal-sized data cache, when caches are large, like the one of the Xeon platform.

The number of L2 misses, for all array sizes, are minimized for $T^2 = C_{L2}$, when the whole cache is been used and a whole tile fits in the cache so that tile-level reuse can be exploited. However, L1 misses are 1 order of magnitude more than L2 misses. As a result, the L1 misses dominate the total memory behaviour, as illustrated in figure 4.

3.4 Data TLB Misses

This cache level is usually fully associative. So, there is no need to take care of array alignment. The TLB miss analysis is similar to the L1 cache analysis. Due to space limitations, we provide only with the corresponding table. Table 2 summarizes the total number of Data TLB misses M_{TLB} for all problem sizes. According to this table, the number of Data TLB misses has the form of figure 3.

The number of TLB misses for all array sizes, for an example of 64 entries (as the TLB size of Xeon is), are minimized when $T = 256$. At this point, the pages addresses of a whole tile fit in the TLB entries, so that tile-level reuse can be exploited.

3.5 Total Miss Cost

Taking into account the miss penalty of each memory level, as well as the penalty of mispredicted branches (as presented in [2]), we derive the total miss cost of

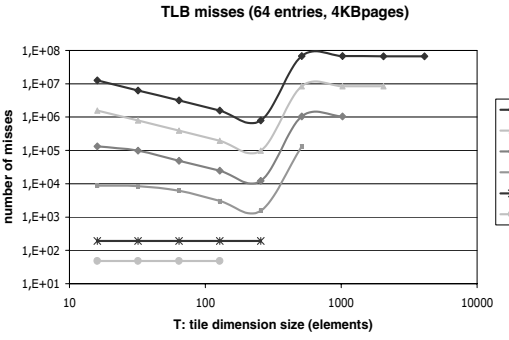

Table 2. D-TLB misses

requirements	M_{TLB}
$N^2 < E \cdot P$	$3\frac{N^2}{P}$
$3TN \leq E \cdot P$	$2\frac{N^2}{P} + \frac{N^3}{T \cdot P}$
$TN < E \cdot P < 3TN$	$\frac{N^2}{P} + 2\frac{N^3}{T \cdot P}$
$T^2 < E \cdot P < 3T^2$	$3\frac{N^3}{T \cdot P}$
$T^2 > E \cdot P > T$	$2\frac{N^3}{T \cdot P} + \frac{N^3}{P}$
$T > E \cdot P$	$\frac{N^3}{T \cdot P} + 2\frac{N^3}{P}$

Fig. 3. Number of TLB misses for various array and tile sizes

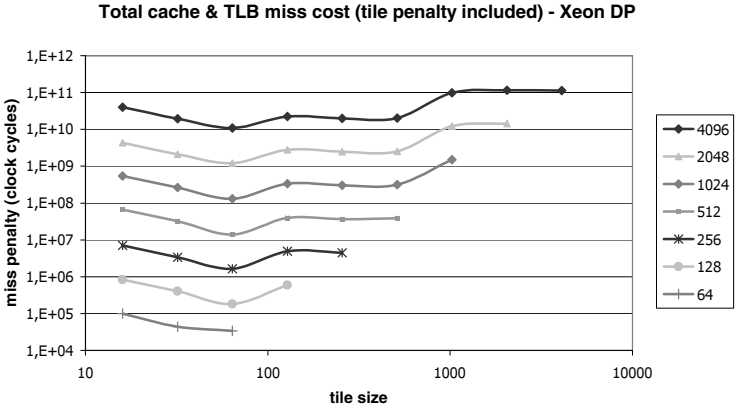

Fig. 4. Number of L2 cache misses on the Xeon DP architecture

figure 4. Figure 4 makes clear that L1 misses dominate cache and, as a result, total performance in the Xeon DP architecture. Maximum performance is achieved when $T = 64$, which is the optimal tile size for L1 cache (the maximum tile that fits in L1 cache). L1 cache misses are more than one order of magnitude more than L2 misses and three orders of magnitude more than TLB misses. Notice that the Xeon DP architecture bears quite a large L2 cache (1Mbytes), which reduces the number of L2 misses significantly, and leaves L1 cache to dominate total performance. Thus, even though L1 misses cost fewer clock cycles, they are still the most weighty factor.

4 Experimental Results

In this section we present experimental results using the Matrix Multiplication benchmarks. The experiments were performed on a Dual SMT Xeon (Simulta-

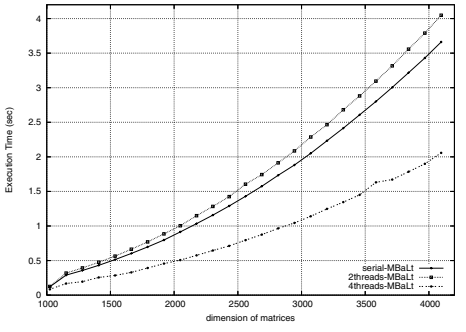

Fig. 5. Xeon - The relative performance of the three different versions

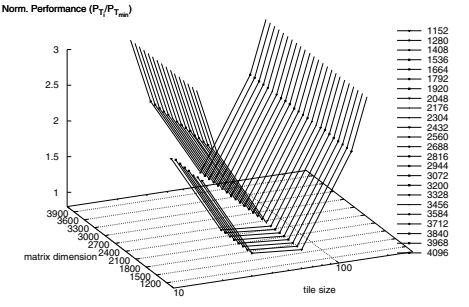

Fig. 6. Xeon - Normalized performance of the matrix multiplication benchmark for various array and tile sizes (serial MBaLt)

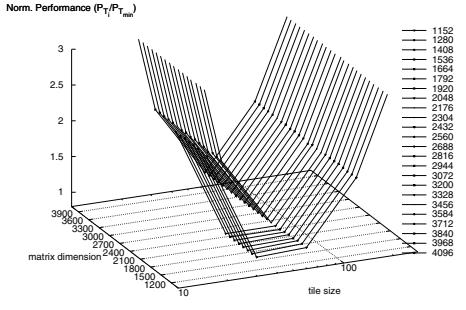

Fig. 7. Xeon - Normalized performance of the matrix multiplication benchmark for various array and tile sizes (2 threads - MBaLt)

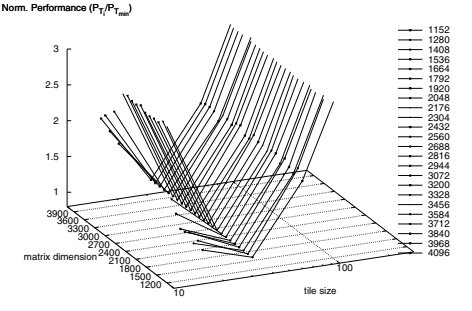

Fig. 8. Xeon - Normalized performance of the matrix multiplication benchmark for various array and tile sizes (4 threads - MBaLt)

neous Multi-Threading supported). The hardware characteristics were described in table 1.

The dual Xeon platform needed special attention, in order to efficiently exploit the hyperthreading technology. We conducted three different experiments. Firstly, the serial blocked algorithm of Matrix Multiplication (MBaLt code - with use of fast indexing) was executed. Secondly, we enabled hyperthreading running 2 threads in the same physical cpu. For large tile sizes, execution times obtained with the 2threads-MBaLt version are quite larger than those of the serial version. Smaller tile sizes lead to more mispredicted branches and loop boundary calculations, thus increasing the overhead of tiling implementation. In the case of 2threads-MBaLt, this tiling overhead gets almost doubled, since the two threads are executing the same code in an interleaved fashion. In other words, the total overhead introduced overlaps the extra benefits we have with the simultaneous execution capabilities of the hyperthreaded processor. This is

not the case for larger tile sizes, where the tiling overhead is not large enough to overlap the advantages of extra parallelism. Figure 5 illustrates only best performance measurements for each different array dimension (tile sizes are equal to the one minimize execution time). The serial-MaLt version seems to have better performance compared to the 2threads-MBaLt version, as execution time is minimized for small tile sizes. Finally, we executed a parallel version of matrix multiplication MBaLt code (4threads-MBaLt), where 2 threads run on each of the 2 physical cpus, that belong to the same SMP. Execution time is reduced, and performance speed up reaches 44% compared to the serial-MBaLt version.

As far as the optimal tile size is concerned, serial MBaLt obey to the general rule, this is $T_{optimal} = \sqrt{C_{L1}} = 64$. However, when hyperthreading had been enabled, $T_{optimal}$ seams to be shifted to the just smaller size, in order to make room in the L1 cache for the increased number of concurrently used array elements. This behaviour is observed, both when two threads run on the same physical cpu (2threads-MBaLt), as well as in the parallel version (4threads-MBaLt) where $T_{optimal} = 32$ (figures 6, 7 and 8). Note that for the 2threads-MBaLt version $T_{optimal} = 32$ when $N < 2048$. For larger arrays, the 2threads-MBaLt version behaves similarly to the serial one, filling the whole L1 cache with useful array elements.

5 Conclusion

A large amount of related work has been devoted to the selection of optimal tile sizes and shapes, for numerical nested loop codes where tiling has been applied. In this paper, we have examined blocked array layouts with an addressing scheme that uses simple binary masks. We have found theoretically and verified through experiments and simulations that, when such layouts are used, in direct mapped caches, with a large L2 cache, L1 cache misses dominate overall performance. Prefetching in combination with other code optimization techniques, set optimal tiling to be $T_1 = \sqrt{C_{L1}}$. On the other hand, when L2 cache has a moderate capacity, L2 misses weight overall performance to larger tile sizes and determine the optimal tile size to be $T \leq \sqrt{C_{L2}}$.

References

1. E. Athanasaki and N. Koziris. Fast Indexing for Blocked Array Layouts to Improve Multi-Level Cache Locality. In *8-th Work. on Interaction between Compilers and Computer Architectures*, Madrid, Spain, Feb 2004. In conjuction with HPCA-10.
2. E. Athanasaki and N. Koziris. A Tile Size Selection Analysis for Blocked Array Layouts. In *9-th Work. on Interaction between Compilers and Computer Architectures*, San Francisco, CA, Feb 2005. In conjuction with HPCA-11.
3. J. Chame and S. Moon. A Tile Selection Algorithm for Data Locality and Cache Interference. In *Int. Conf. on Supercomputing*, Rhodes, Greece, June 1999.
4. S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Conf. on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

5. K. Esseghir. Improving Data Locality for Caches. Master's thesis, Department of Computer Science, Rice University, Houston, TX, Sept 1993.
6. S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Trans. on Programming Languages and Systems*, 21(4), July 1999.
7. J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical Modeling of Set-Associative Cache Behavior. *IEEE Trans. Computers*, 48(10), Oct 1999.
8. C.-H. Hsu and U. Kremer. A Quantitative Analysis of Tile Size Selection Algprithms. *The J. of Supercomputing*, 27(3), Mar 2004.
9. M. Kandemir, J. Ramanujam, and A. Choudhary. Improving Cache Locality by a Combinaion of Loop and Data Transformations. *IEEE Trans. on Computers*, 48(2), Feb 1999.
10. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.
11. K. S. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. on Programming Languages and Systems*, 18(04), July 1996.
12. N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the Multi-Level Nature of Tiling Interactions. *Int. J. of Parallel Programming*, 26(6), Dec 1998.
13. P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. *IEEE Trans. on Computers*, 48(2), Feb 1999.
14. N. Park, B. Hong, and V. Prasanna. Analysis of Memory Hierarchy Performance of Block Data Layout. In *Int. Conf. on Parallel Processing*, Vancouver, Canada, Aug 2002.
15. D. Patterson and J.Hennessy. *Computer Architecture. A Quantitative Approach.* San Francisco, CA, 3rd edition, 2002.
16. G. Rivera and C.-W. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Int. Conf. on Supercomputing*, Melbourne, Australia, July 1998.
17. G. Rivera and C.-W. Tseng. A Comparison of Compiler Tiling Algorithms. In *Int. Conf. on Compiler Construction*, Amsterdam, The Netherlands, March 1999.
18. G. Rivera and C.-W. Tseng. Locality Optimizations for Multi-Level Caches. In *Int. Conf. on Supercomputing*, Portland, OR, Nov 1999.
19. Y. Song and Z. Li. Impact of Tile-Size Selection for Skewed Tiling. In *5-th Work. on Interaction between Compilers and Architectures*, Monterrey, Mexico, Jan 2001.
20. O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomena. In *Conf. on Measurement and Modeling of Computer Systems*, Nashville, TN, May 1994.
21. O. Temam, E. D. Granston, and W. Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Conf. on Supercomputing*, Portland, OR, Nov 1993.
22. X. Vera. *Cache and Compiler Interaction (how to analyze, optimize and time cache behaviour).* PhD thesis, Malardalen University, Jan 2003.
23. M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Conf. on Programming Language Design and Implementation*, Toronto, Canada, June 1991.
24. M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining Loop Transformations Considering Caches and Scheduling. In *Int. Symposium on Microarchitecture*, Paris, France, Dec 1996.