John von Neumann Institute for Computing

Reducing Cache Misses by Loop Reordering

E. Herruzo, G. Bandera, E.L. Zapata, O. Plata

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 541-548, 2006.

http://www.fz-juelich.de/nic-series/volume33

Reducing Cache Misses by Loop Reordering

E. Herruzo[a], G. Bandera[b], E.L. Zapata[b], O. Plata[b]

[a]Department of Electronics, University of Córdoba, Spain

[b]Department of Computer Architecture, University of Málaga, Spain

This paper presents a novel method to determine the best loop reordering of a perfect loop nest with the aim of maximizing the resulting cache set occupation. The method is based on a simplified analytical model of the cache, reducing the cache behaviour of interest by a small number of parameters. These parameters contain the meaning of the probability of self-interference in the cache due to memory references to a particular array. Based in such parameters, we have designed a fast and effective algorithm to reorder (permute) the loops in the nest so that self-interferences in cache due to references to a particular array is minimized. An evaluation of the method is also presented.

1. Introduction

During the last decades speed of processors has been widely improved, however memory speed could not keep pace. Memory hierarchies, and specifically cache memories, were introduced to solve the performance penalty related to the speed gap. Memory hierarchy introduces the latency problem to access data. The memory latency has been attacked from two different fronts. On the one hand, by means of hardware solutions, like lockup-free caches, prefetching, out-of-order execution, and so on. On the other hand, compiler techniques have been developed to fully make use of the available hardware structures. The efficiency of architectural improvements depends on the ability of the compiler to change the structure of programs for taking full advantage of them. The most important compiler optimizations are basically loop and data layout transformations.

We present in this paper a new method to determine the best loop permutation of a perfect loop nest which tries to maximize the cache set occupation. There are several algorithms for loop permutation in the literature [2,6,9,11], being the *Loopcost* algorithm [4,1] one of the best known. Our loop permutation method is based on a simplified analytical cache model, resulting in a small set of parameters to determine the cache behaviour we are interested in. *Loopcost* shares some similar basic principles, but our method has significant differences both in the implementation and in the cache model, so as it provides a better optimization of the cache use for many important cases.

The rest of the paper is organized as follows. Section 2 presents the simplified analytical cache model. The algorithm we propose for loop permutation is introduced in Section 3. Section 4 presents some related work, and Section 5 shows the experimental evaluation of our method, compared to *Loopcost* (and a commercial compiler). Finally, section 6 draws some conclusions.

2. Modelling the Cache Behaviour

We present in this section a simplified model of the cache behaviour when specific data access patterns to memory originate from the execution of a loop nest. The aim is to define a number of parameters that characterize such a cache behaviour and could be used to determine the most suitable loop reordering, so as cache set occupation is maximized.

In order to state the model, we will consider a $k-$way set-associative cache, of size $C \times L \times W$ (C is the number of cache sets, L is the number of blocks per set, and W is the block size in words).

We also consider a $M-$dimensional array X, stored in a column-major order, which is referenced within a perfectly nested loop.

Without loss of generality, we assume that array references are made inside a $M-$depth nested loop with iteration vector $\vec{I} = (I_1, I_2, \ldots, I_M)$. Expressions in the array dimensions are of the form $f_k * I_k$, $k = 1, \ldots, M$, but any general affine expression is perfectly valid.

When the multidimensional array X is allocated in memory, it is linearized and laid out in some order. We can obtain the reference in memory to X for each iteration of the loop nest by means of a LMAD (Linear Memory Access Descriptor) [14]. So, the memory reference due to the access to X in some iteration $\vec{I} = (I_1, I_2, \ldots, I_M)$ of the nested loop is as shown (a column-major order is assumed),

$$MemRef(X, \vec{I}) = f_1 * I_1 + \ldots + f_k * I_k * \prod_{i=1}^{k-1} D_i + \ldots + f_M * I_M * \prod_{i=1}^{M-1} D_i, \quad (1)$$

where D_i is the size of the $i-$dimension of X.

The *stride* of array X on loop index I_k is defined as the distance in memory of array entries referenced by consecutive iterations of loop k, that is,

$$Stride(X, I_k) = f_k * I_k^{l+1} * \prod_{i=1}^{k-1} D_i - f_k * I_k^l * \prod_{i=1}^{k-1} D_i, \quad (2)$$

where I_k^l represents the $l-$th iteration of the loop k.

To simplify the explanation, we restrict the analysis of the cache behaviour to a single array X inside a perfectly nested loop. However this analysis can be easily extended to several arrays appearing in the same loop and to not-perfectly nested loops. The goal of our analysis is to describe and represent array self-interferences in cache due to the execution of the loop nest. That is, to determine how memory blocks with data from X are located in the cache and may replace other previously placed blocks with data from the same array.

As a first step of our study we must define a number of cache parameters. They will be used afterwards to define the proposed reordering method. During the execution of the loop, blocks of data from main memory are located in the cache. These blocks are mapped to the cache sets as shown in the following equation,

$$Set(X, \vec{I}) = \Big\lfloor \frac{MemRef(X, \vec{I})}{W} \Big\rfloor \bmod C. \quad (3)$$

The execution of consecutive iterations of loop k generates memory references separated a distance of $Stride(X, I_k)$. The distance (in cache sets) of the blocks containing these referenced data, once they are placed in cache, is a some sort of a cache *set stride*, defined as,

$$SetStride(X, I_k) = \frac{Stride(X, I_k)}{W}. \quad (4)$$

There is no module operation in the above expression because we are considering only the linear placement of cache blocks (from the beginning to the end of the cache). From this point on, the rest of blocks are placed starting again from the beginning of the cache and may produce self-replacement (that is, replacement of blocks containing X data previously placed). And this process

may be repeated several times (until all iterations of loop k are exhausted). Note also that during this first linear placement of cache blocks, each block is allocated in a different cache set.

Now, we can calculate the total number of memory blocks that can be placed in the cache with no opportunity of self-replacement (by dividing the total number of sets in the cache by the cache *set stride*). After that, the rest of the iterations of loop k may produce self-replacement. On the other hand, we can easily calculate the total number of memory blocks occupied by all the array entries of X that are referenced during the whole execution of loop k. All these blocks must be placed in the cache. Dividing this value by the number of blocks that fits the cache in a linear placement, we obtain a parameter which approximates the number of linear block placements in the cache due to references to X during the execution of the loop k. This parameter is denoted by $CacheTurns(X, I_k)$ and is calculated as follows,

$$CacheTurns(X, I_k) = \frac{N_k * Stride(X, I_k) * SetStride(X, I_k)}{C * W}, \tag{5}$$

where N_k is the number of iterations of loop k. $CacheTurns(X, I_k)$ depends on the stride in memory of array X in loop k, the number of iterations of such loop, and the cache properties. It can be seen that if $CacheTurns(X, I_k)$ is less or equal to one, then all the references to array X in loop k fit the cache with no self-replacements. Otherwise, if such parameter is greater than 1, a probability of self-replacement exists. In fact, higher values of it represent higher opportunities of self-replacement.

The $CacheTurns()$ parameter gives some rough information about the miss behaviour of the cache for some specific memory access patterns. We show in this paper that this information is enough to decide how to arrange the loops in the nest in order to reduce the probability of cache misses due to self-interferences.

In order to obtain a simple reordering algorithm, we disregarded two important issues that, however, do not influence much the effectiveness of our method. On the one hand, the fact of not taking into account the possible interference with other arrays used in the same loop nest. On the other hand, the fact that there is not always a self-replacement on the array elements already contained in the cache (set associativity).

3. Loop Permutation

We present in this section an algorithm to decide the loop arrangement (permutation) that minimizes the cache miss rate due to array self-interferences. As explained previously, a high value of $CacheTurns()$ implies a high probability of cache block replacement due to self-interference. Taking this fact into account, our permutation algorithm looks for loops obtaining higher values of $CacheTurns()$ and places them in the outermost position of the loop nest. As a result, loops with small values of $CacheTurns()$ will be placed in the innermost positions. This way, cache blocks are re-used before having the opportunity to be replaced by self-interference. Fig. 1 outlines the permutation algorithm, that we call *TCacheTurns*.

In contrast to other authors, our permutation algorithm relates not only the number of loop iterations to the characteristic parameters of the cache but it also takes into account how the data inside the nested loop is referenced (stride). Note also that our algorithm agrees with the fact that, in general, it is better to have the stride-1 array references in the innermost loop of the nest in order to exploit spatial locality [3,1,4].

for (each array X) **do**
  for (each loop I_k) **do**
    Compute $Stride(X, I_k)$ and $SetStride(X, I_k)$.
    if (I_k is in $MemRef(X, I)$) **then** Compute $CacheTurns(X, I_k)$.
    else $CacheTurns(X, I_k) = 0$.
  end for
end for
for (each I_k) **do**
  for (each array X) **do**
    Add $CacheTurns(X, I_k)$ to $TotalCacheTurns(I_k)$.
  end for
endfor
Place loops in the nest in decreasing order of $TotalCacheTurns(I_k)$.

Figure 1. Loop permutation algorithm (*TCacheTurns*)

4. Related Work

Some authors attempt to unify loop and data layout transformations. Kandemir et al. [10] present a method that uses ILP (Integer Linear Programming) to calculate optimal solutions for data transformations. Their paper describes an approach to detect the data layout of different arrays in memory, together with the best loop permutation for each loop nest in the source code. Although they include an iterative method to calculate memory layout transformations, they do not indicate any precise algorithm for loop permutation. We have found other works describing an heuristic solution to both data layout and loop transformations [13], but neither of them present algorithms to determine the code transformation.

Ghosh et al. [8] describe an approach to calculate the Cache Miss Equations (CME). Their algorithm uses the reference reuse vector, instead of using the stride. One of the main problems of this approach is the expensive process to calculate the miss equations. D'Alberto et al. [7] present a static analysis of parameterized loop nests. It is based on CMEs and on static cache parameters. They use this analysis to detect the interferences or cache block replacements of the same/distinct array references.

The work presented by Clauss and Meister [5] is mainly focused on spatial locality optimization. Their method consists in providing a new array reference function to the compiler. This is a parameterized cost function based on polytopes and Ehrhart polynomials from the iteration space of a loop nest. Clauss et al. [6] and Loechner et al. [12] use the same framework to define optimization techniques for the TLB. In this paper, we also work with the polyhedron defined by the iteration space of a loop nest. We need to know the data access layout (or polyhedron structure) to determine the cache memory occupation for each loop.

The Data Relation Vectors are defined by Kandemir and Ramanujam [11] to describe a framework to establish some compiler optimizations to improve data reuse. Weikle et al. [15] use these vectors and a mathematical model of the cache to establish a formal notation that enables compiler optimizations.

Carr et al. [4] define the *loopcost* algorithm, that is used by Allen and Kennedy [1] to develop

Table 1
L2 data cache misses for different loop counts (multiplication of 800×800 matrices)

M	N	P	Algorithm	Loop Arrangement	L2 Misses
400	400	400	Loopcost	j, k, i	6925
			TCacheTurns	j, k, i	6925
400	800	800	Loopcost	j, k, i	33910
			TCacheTurns	j, k, i	33910
400	400	800	Loopcost	j, k, i	28870
			TCacheTurns	k, j, i	16180
400	800	400	Loopcost	j, k, i	9570
			TCacheTurns	j, k, i	9570
800	400	800	Loopcost	j, k, i	110520
			TCacheTurns	k, j, i	24270
800	400	400	Loopcost	j, k, i	11210
			TCacheTurns	j, k, i	11210
800	400	200	Loopcost	j, k, i	5610
			TCacheTurns	j, k, i	5610
800	200	400	Loopcost	j, k, i	8070
			TCacheTurns	k, j, i	6510

a loop permutation technique which, in a particular way, is similar to the one described here but with some important differences. Usually the loop permutation resulted from both algorithms is the same, but we obtain a different one when the array is multi-dimensional and there are several loop indices in the non-contiguous dimension. The result is also different when the loops in the nest have different sizes. The main difference between both algorithms lies in how is calculated the cost for every reference within the innermost loop. The goal of the *loopcost* algorithm is to set the loop that occupies less cache sets as the innermost loop. The *loopcost* approach classifies array references into groups that exhibit temporal or spatial reuse. In our case, we take into account the stride defined by the LMAD of every reference, and we only join references when they exhibit spatial-group reuse. In the section about experimental results we compare both algorithms.

5. Experimental Results

This section evaluates our permutation algorithm *TCacheTurns* and compare it with *loopcost* and a real commercial compiler. Our experiments were conducted in a platform with a R10k processor running in an exclusive mode. We have used the MIPSPro Fortran90 compiler (version 7.30) with $-O0$ compiler optimization option. The hardware counters of the processor have been tested using *Perfex functions*. We carried out two sets of experiments, comparing L2 data cache misses. The first set of tests is based in the matrix-matrix multiplication problem, while the second one compares the different solutions for several benchmarks extracted from NAS, PerfectB and SPEC2000.

5.1. The matrix-matrix multiplication problem

We accomplished two different types of tests: the first one analyzes the effect in L2 cache miss rate of different loop counts in the matrix multiplication problem with fixed-size input matrices (all three matrices are of size 800×800). The second one changes the size of the input matrices but

Table 2
L2 data cache misses for different matrix dimensions (loop counts $M = N = P = 1000$)

X matrix dim.	Y matrix dim.	Z matrix dim.	Algorithm	Loop Arrangement	L2 Misses
1000,1000	1000,1000	1000,1000	Loopcost	j, k, i	1.2 Mill
			TCacheTurns	j, k, i	1.2 Mill
1000,1000	1000,1000	3000,3000	Loopcost	j, k, i	1.3 Mill
			TCacheTurns	j, k, i	1.3 Mill
1000,1000	3000,3000	1000,1000	Loopcost	j, k, i	1.5 Mill
			TCacheTurns	j, k, i	1.5 Mill
1000,1000	3000,3000	3000,3000	Loopcost	j, k, i	1.6 Mill
			TCacheTurns	j, k, i	1.6 Mill
3000,3000	1000,1000	1000,1000	Loopcost	j, k, i	9.6 Mill
			TCacheTurns	k, j, i	4.2 Mill
3000,3000	1000,1000	3000,3000	Loopcost	j, k, i	9.6 Mill
			TCacheTurns	j, k, i	9.6 Mill
3000,3000	3000,3000	1000,1000	Loopcost	j, k, i	9.7 Mill
			TCacheTurns	j, k, i	9.7 Mill

keeping the loop counts. Table 1 shows the experimental results for the first type of tests. There are three cases where the loop arrangement resulted from *loopcost* and *TCacheTurns* algorithms are different. That occurs when the number of iterations of loop k is much larger than that of loop j. In all cases our algorithm obtains a lower number of L2 data cache misses.

Table 2 shows a comparison between *loopcost* and *TCacheTurns* for the second type of tests. All loops are of the same size, a total of 1000 iterations each one. This set of experiments shows that normally the loop arrangement and the number of L2 cache misses resulting from applying both algorithms is the same. However, our approach obtains a different loop reordering with a lower number of cache misses for some cases. This occurs when the size of the first matrix (X) is greater than the size of the other two. In this situation, as the k loop iterates over the non-contiguous dimension of the X matrix, the corresponding stride ($Stride(X, k)$) will be greater than for the other two matrices Y and Z. The size of X is large enough to produce a different loop arrangement and, consequently, a better performance.

5.2. NAS, PerfectB and SPEC2000 benchmarks

A comparison of the different considered solutions using a selection of various benchmarks is shown in Table 3. It can be seen that, for multidimensional arrays, our algorithm outperforms the other two approaches. In the rest of cases, the number of L2 cache misses obtained is the same. The reason for this behaviour comes from the use of the stride parameter in the determination of the best loop permutation. *Loopcost* only uses the number of iterations and considers the same cost for loops traversing non-contiguous dimensions. However, this different way of compute the best loop arrangement does not make any difference for 2D arrays with the same number of iterations, as it is the case of `fftpde` and `swim`. Some cases in which *TCacheTurns* obtains a better loop permutation than *loopcost* occur when the nest contains loops accessing non-contiguous dimensions of multi-dimensional arrays. Together with the above two algorithms we include in this table the results obtained with the native compiler with the loop permutation option switched on ($-LNO$).

Table 3
Benchmark subroutines, running conditions and L2 data cache misses

PerfectB B.	Subrout.	Loop counts	Array dim.	Algorithm	Loop Arrangem.	L2 Misses
adm	hyd	100,100,100	100,100,100	Loopcost	j,k,i	64.680
				TCacheTurns	k,j,i	19.260
				Compiler	compiler selection	64.680
flo52	collc	200,200,200	200,200,200	Loopcost	j,n,i	351.660
				TCacheTurns	n,j,i	3.450
				Compiler	compiler selection	351.660
dyfesm	mnlbyx	500,500,500	500,500	Loopcost	j,n,i	203.524
				TCacheTurns	n,j,i	37.080
				Compiler	compiler selection	162.668
migration	migrat	200,200,200	200,200,2,200	Loopcost	j,k,i	358.920
				TCacheTurns	k,j,i	4.100
				Compiler	compiler selection	358.920
NAS Benchm.						
appbt	l2norm	48,48,48,5	5,50,50,50	Loopcost	i,j,k,m	2.550
				TCacheTurns	k,j,i,m	34
				Compiler	compiler selection	2.560
appsp	spentax3	30,30,600	660,33,33	Loopcost	j,k,i	89.850
				TCacheTurns	k,j,i	3.340
				Compiler	compiler selection	89.230
fftpde	transx	500,500	1000,1000	Loopcost	i,j	4.500
				TCacheTurns	i,j	4.500
				Compiler	compiler selection	4.500
SPEC2000 B.						
gangel	polnel	80,80,80	100,100,100	Loopcost	j,l,i	158.270
				TCacheTurns	l,j,i	73.970
				Compiler	compiler selection	158.270
applu	rhs	45,45,45,5	5,50,50,50	Loopcost	j,k,i,l	30.100
				TCacheTurns	k,j,i,l	3.250
				Compiler	compiler selection	30.100
mgrid	psinv	150,150,150	150,150,150	Loopcost	k,j,i	302.340
				TCacheTurns	j,k,i	4.150
				Compiler	compiler selection	302.340
swim	calc3	1000,1000	1335,1335	Loopcost	j,i	29.600
				TCacheTurns	j,i	29.600
				Compiler	compiler selection	29.600

The results with the native compiler are also worse, in many cases, than our algorithm.

6. Conclusions

In this paper we presented an algorithm to determine how to arrange the loops in a nest so as the number of cache misses is minimized due to array self-interferences. It is based on a novel model of the cache, where parameters were defined to give information about the cache memory occupation

when affine array references to main memory are issued from a nested loop. Our algorithm (*TCacheTurns*) was compared with the well-known *loopcost* algorithm, obtaining similar or better results. We have shown how a simple model of a cache allows to determine a very effective arrangement of loops in a nest that optimizes the occupation of the cache. As a future work we will study how to define a general formal framework to apply a set of important compiler optimization techniques, like padding, tiling, loop reversal, and so on.

References

[1] J. R. Allen and K. Kennedy: *Optimizing Compilers for Modern Architectures: A Dependece-based Approach*. **Morgan Kaufmann Publishers**. October 2001.

[2] D.F. Bacon, S.L. Graham and O.J. Sharp: *Compiler Transformations for High-Performance Computing*. **ACM Computing Surveys**, Vol. 26, No. 4, pp. 345-420. December 1994.

[3] D.H. Bailey: *Unfavorable Strides in Cache Memory Systems*. **Technical Report RNR-92-015**, NASA Ames Research Center, CA. 1992.

[4] S. Carr, K.S. Mckinley and C. Tseng: *Compiler Optimizations for Improving Data Locality*. **In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems**, San Jose, CA. October 1994.

[5] Ph. Clauss and B. Meister: *Automatic Memory Layout Transformation to Optimize Spatial Locality in Parameterized Loop Nests*. **ACM SIGARCH Computer Architecture News**, Vol. 28, No. 1. March 2000.

[6] Ph. Clauss, V. Loechner and B. Meister: *Minimizing strides in Loops with Affine Array References*. **In Proceedings of the Compilers for Parallel Computers**, Edinburgh, Scotland. June 2001.

[7] P. D'Alberto, A. Nicolau, A. Veidenbaum and R. Gupta: *Static Analysis of Parameterized Loop Nests for Energy Efficient Use of Data Caches*. **Compilers and Operating Systems for Low Power**, pp. 193-207, Norwell, MA, USA, Kluwer Academic Pub. 2003.

[8] S. Ghosh, M. Martonosi and S. Malik: *Cache Miss Equation: An Analytical Representation of Cache Misses*. **In Proceedings of the 11th International Conference on Supercomputing**, Vienna, Austria. 1997.

[9] S. Ghosh, M. Martonosi and S. Malik: *Cache Miss Equations: A Compiler Framework for Analyzing and Tunning Memory Behaviour*. **ACM Transactions on Programming Language and Systems**. July 1999.

[10] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam and E. Ayguade: *An Integer Linear Programming Approach for Optimizing Cache Locality*. **In Proceedings of the 13th International Conference on Supercomputing**, Rhodes, Greece. June 1999.

[11] M. Kandemir and J. Ramanujam: *Data Relaton Vectors: A New Abstraction for Data Optimizations*. **IEEE Transactions on Computers**, Vol. 50, No. 8. August 2001.

[12] V. Loechner, B. Meister and Ph. Clauss: *Precise Data Locality Optimization of Nested Loops*. **The journal of Supercomputing**, Vol. 21, No. 1, pp. 37-76, Kluwer Academic Pub. 2002.

[13] M. O'Boyle and P. Knijnenburg: *Integrating Loop and Data Transformations for Global Optimizations*. **IEEE Int'l. Conf. on Parallel Architectures and Compilation Techniques**, Paris, France. October 1998.

[14] Y. Paek, J. Hoeflinger and D. Padua: *Simplification of Array Access Patterns for Compiler Optimizations*. **In Proceedings of the SIGPLAN Conference of Programming Language Design and Implementation**. June 1994.

[15] A.D.B. Weikle, S.A. McKee, K. Skadrom and W.A. Wulf: *Cache As Filters: A Framework for the Analysis of Caching System*. **In Proceedings of th Grace Murray Hopper Conference 2000**. September 2000.