Advanced Data Layout Optimization for Multimedia Applications

Chidamber Kulkarni[1,2] Francky Catthoor[1,3] Hugo De Man[1,3]

[1] IMEC, Kapeldreef 75, B3001 Leuven, Belgium.
[2] Also Ph.D. student at the Dept of EE, Kath Univ Leuven
[3] Also Professor at the Dept of EE, Kath Univ Leuven
<kulkarni,catthoor,deman@imec.be>

1 Introduction and Related Work

Increasing disparity between processor and memory speeds has been a motivation for designing systems with deep memory hierarchies. Most data-dominated multimedia applications do not use their cache efficiently and spend much of their time waiting for memory accesses [1]. This also implies a significant additional cost in increased memory bandwidth, in the system bus load and the associated power consumption apart from increasing the average memory access time.

In this work, we are mainly targeting the embedded (parallel) real-time multimedia processing (RMP) application domain since algorithms in there lend themselves to very good compile-time analysis. Although embedded RMP applications are relatively regular, but certainly not perfectly linear/affine in the loop and index expressions, the simultaneous presence of complex accesses to large working sets makes most of the existing approaches largely to fail in taking advantage of the locality. Earlier studies have shown that the majority of the execution time is spent in cache stalls due to cache misses for image processing applications [1] as well as scientific applications [12]. Hence the reduction of such cache misses is of crucial importance.

Source-level program transformations to modify the execution order can improve the cache performance of these applications to a large extent [3, 6–9] but still a significant amount of cache misses are present. Storage order optimizations [3, 4] are very helpful in reducing the capacity misses. So in the end mostly conflict cache misses related to the sub-optimal data layout remain. Array padding has been proposed earlier to reduce the latter [11, 14, 15]. These approaches are useful for reducing the cross-conflict misses. However existing approaches do not eliminate the majority of the conflict misses. Besides [2, 6, 14], very little has been done to measure the impact of data organization (or layout) on the cache performance. Thus there is a need to investigate additional data layout or organization techniques to reduce these cache misses.

The fundamental relation which governs the mapping of data from the main memory to a cache is given as below :

$$(Block\ Address)\ MOD\ (Number\ of\ Sets\ in\ Cache) \tag{1}$$

Based on the number of lines in a set we define direct mapped, n-way associative and fully associative cache [16]. It is clear that, if we arrange the data in the main memory so that they are placed at particular block addresses depending on their lifetimes and sizes, we can control the mapping of data to the cache and hence (largely) remove the influence of associativity on the mapping of data to the cache. The problem is however that trade-offs normally need to be made between many different variables. This requires a global data layout approach which to our knowledge has not yet been published before. This has been the motivation for us to come up with a new formalized and automated methodology for optimized data organization in the higher levels of memory. Our approach is called main memory data layout organization (MDO). This is our main contribution, which will be demonstrated on real-life applications.

The remaining paper is organized as follows : Section 2 presents an example illustration of the proposed main memory data layout organization methodology. This is followed by the introduction of the general memory data layout organization problem and the potential solutions in section 3. Experimental results on three real-life test vehicles are presented in section 4. Some conclusions from this work are given in section 5.

2 Example Illustration

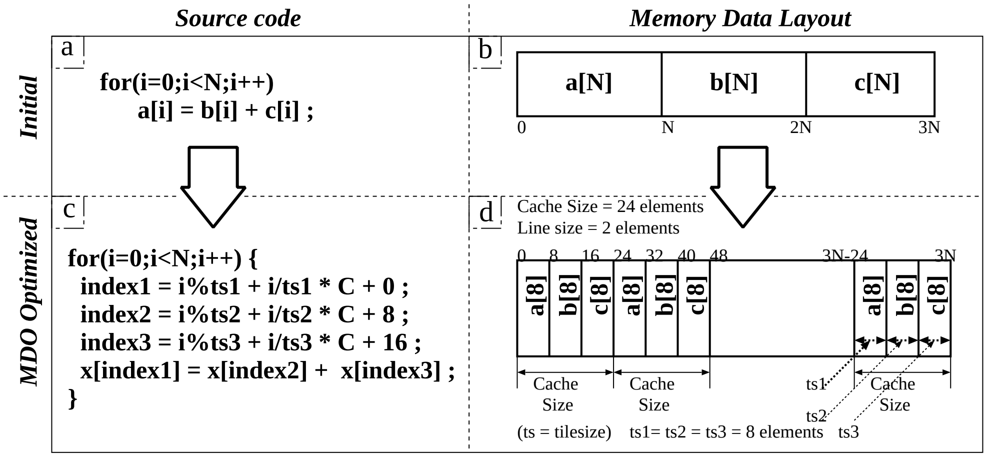
In this section we will briefly introduce the basic principle behind main memory data layout organization (MDO) using an example illustration.

Consider the example in figure 1. The initial algorithm in figure 1(a) needs three arrays to execute the complete program. Note that the initial main memory data layout in figure 1(b) is single contiguous irrespective of the array and cache sizes. The initial algorithm can have $3N$ (cross-) conflict cache misses for a direct mapped cache, in the worst case i.e. when each of the arrays are placed at an (initial) address, which is a multiple of the cache size. Thus to eliminate all the conflict cache misses, it is necessary that none of the three arrays gets mapped to the same cache locations.

The MDO optimized algorithm, as shown in figure 1(c), will have no (cross-) conflict cache misses at all. This is because, in the MDO optimized algorithm the arrays always get mapped to fixed and non-overlapping locations in the cache. This happens because of the way the data is stored in the main memory, as shown in figure 1(d). To obtain this modified data layout, the following steps are carried out :

1. the initial arrays are split into sub-arrays of equal size. The size of each sub-array is called the *tile size*.
2. different arrays are merged so that the sum of their tile-sizes equals the cache size. Now store the merged arrays recursively till all the concerned arrays are completely mapped in the main memory. Thus we now have a new array which comprises all the arrays but the constituent arrays are stored in such a way that they get mapped into cache so as to remove the conflict misses. This new array is represented by “x[]” in figure 1(c).

In figure 1(c) and (d), two important observations need to be made : (1) there is a recursive allocation of different array data, with each recursion equal to the cache size and (2) the generated addressing, which is used to impose the modified data layout on the linker, contains modulo operations. These can be removed afterwards through a seperate optimization stage [5].



```
for(i=0;i<N;i++)
    a[i] = b[i] + c[i] ;
```

```
for(i=0;i<N;i++) {
  index1 = i%ts1 + i/ts1 * C + 0 ;
  index2 = i%ts2 + i/ts2 * C + 8 ;
  index3 = i%ts3 + i/ts3 * C + 16 ;
  x[index1] = x[index2] +  x[index3] ;
}
```

Fig. 1. Example illustration of MDO optimization on a simple case.

3 Main Memory Data Layout Organization (MDO)

In this section, we will first present a complete problem formulation invovling the two stages namely the tile size evaluation and the array merging as introduced in section 2. To deal with complex realistic applications the optimal solution would require too much CPU time. So we have also developed a heuristic, which has been automated in a source-to-source precompiler step.

3.1 The General Problem

The general main memory data layout organization problem for efficient cache utilization (DOECU) can be stated as,"For a given program with m-loop nests and n-variables (arrays), obtain a data layout which has the least possible conflict misses". This problem has two sub problems. First, the tile size evaluation problem and secondly the array merging/clustering problem.

Tile Size Evaluation Problem : Let x_i be the tile size of the array i and C be the cache size. For a given program we need to solve the m equations as below to obtain the needed (optimal) tile sizes. This is required because of two reasons. Firstly, an array can have different effective size in different loop nests. We define effective size as "the number of elements of an array accessed in a loop nest". This number can thus represent either the complete array size or a partial size and is represented as *effsize.* The second reason being that different loop nests have different number of arrays which are simultaneously alive.

$$L_1 = x_1 + x_2 + x_3 + ... + x_n = C \tag{2}$$

$$L_2 = {x_1}^1 + {x_2}^1 + {x_3}^1 + ... + {x_n}^1 = C \tag{3}$$

$$. \quad . \quad . \quad .$$

$$L_m = {x_1}^{(m-1)} + {x_2}^{(m-1)} + {x_3}^{(m-1)} + ... + {x_n}^{(m-1)} = C \tag{4}$$

The above equations need to be solved so as to minimize the number of conflict misses. In this paper, we assume that all the arrays which are simultaneously alive have an equal probability to conflict (in the cache). The optimal solution to this problem comprises solving ILP problem [10, 13], which requires large CPU time. Hence we have developed heuristics which provide good results in a reasonable CPU time.

Array Merging/Clustering Problem : We now further formulate the general problem using the loop weights. The weight in this context is the probability of conflict misses calculated based on the simultaneous existence of arrays for a particular loop-nest i.e. sum of effective sizes of all the arrays as given below :

$$L_{wk} = \sum_{i=1}^{n} effsize_i \tag{5}$$

Hence, now the problem to be solved is, which variables to be clustered or merged and in what order i.e. from which loop-nest onwards so as minimize the cost function. Note that we have to formulate the array merging problem this way because, we have many tile sizes for each array[1] and there are different number of arrays alive in different loop nest. Thus, using above loop weights we can identify loop nests which can potentially have more conflict misses and focus on clustering arrays in these loop nests.

3.2 The Pragmatic Solution

We now discuss some pragmatic solutions for the above problem. These solutions comprise heuristics, which are less complex and faster from the point of view of automation. First, we briefly discuss how the two stages of the problem are solved as below :

1. The first step involves evaluation of the effective sizes for each array instances in the program. Next, we perform a proportionate allocation based on the effective size of every array in every loop nest. This means that arrays with

[1] In the worst case, one tile size for every loop nest in which the array is alive.

larger effective sizes get larger tile sizes and vice-versa. Thus the remaining problem is the merging of different arrays.

2. The second step involves the merging/clustering of different arrays with their tile sizes. To achieve this we first arrange all the loop nests (in our internal model), in ascending order of their loop weights as calculated earlier. Next, we start merging arrays from the loop nest with highest loop weight and go on till the last remaining array has been merged. Note that once the total tile size is equal to the cache size, we start a second cluster and so on. This is done in a relatively greedy way, since we do not explore for the best possible solution extensively.

We have automated two heuristics in a prototype tool, which is a source-to-source (C-to-C) pre-compiler step. The basic principle of these two heuristics are given below :

1. *DOECU I :* In the first heuristic, the tile size is evaluated individually for each loop nest i.e. the proportionate allocation is performed based on the effective sizes of each array in the particular loop nest itself. Thus we have many alternatives[2] for choosing the tile size for an array. In the next step, we start merging the arrays from the loop nest with the highest weight, as calculated earlier, and move to the loop nest with the next highest weight and so on till all the arrays are merged. In summary, we evaluate the tile sizes locally but perform the merging globally based on loop weights.
2. *DOECU II :* In the second heuristic, the tile sizes are evaluated by a more global method. Here we first accumulate the effective sizes for every array over all the loop nests. Next we perform the proportionate allocation for every loop nest based on the accumulated effective sizes. This results in lesser difference between tile size evaluated for an array in one loop nest compared to the one in another loop nest. This is necessary since sub-optimal tile sizes can result in larger self conflict misses. The merging of different arrays is done in a similar was as in the first heuristic.

4 Experimental Results

This section presents the experimental results of applying MDO, using the prototype DOECU tool, on three different real-life test-vehicles namely a cavity detection algorithm used in medical imaging, a voice coder algorithm which is widely used in speech processing and a motion estimation algorithm used commonly in video processing applications. Note that all three algorithms are quite large and due to limitations in space we will not explain the algorithmic details of these applications[3].

[2] We could have in the worst case, a different tile size for every array in every loop nest for the given program.

[3] In brief, cavity detection algorithm is 8 pages and has 10 loopnests, voice coder algorithm is 12 pages and has 22 loopnests and motion estimation is 2 pages and has one loopnest with a depth of six.

The initial C source code is transformed using the prototype DOECU tool, which also generates back the transformed C code. These two C codes, initial and MDO optimized, are then compiled and executed on the Origin 2000 machine and the performance monitoring tool "perfex" is used to read the hardware counters on the MIPS R10000 processor.

Table 1, table 2 and table 3 show the obtained results for the different measures for all the three applications. Note that table 3 has same result for both the heuristics since the motion estimation algorithm has only one (large) loop nest with a depth of six namely six nested loops with one body.

	Initial	DOECU (I)	DOECU (II)
Avg memory access time	0.482180	0.203100	0.187943
L1 Cache Line Reuse	423.219241	481.172092	471.098536
L2 Cache Line Reuse	4.960771	16.655451	23.198864
L1 Data Cache Hit Rate	0.997643	0.997926	0.997882
L2 Data Cache Hit Rate	0.832236	0.943360	0.958676
L1–L2 bandwidth (MB/s)	13.580039	4.828789	4.697513
Memory bandwidth (MB/s)	8.781437	1.017692	0.776886
Actual Data transferred L1–L2 (in MB)	6.94	4.02	3.70
Actual Data transferred L2–Memory (in MB)	4.48	0.84	0.61

Table 1. Experimental Results for the cavity detection algorithm using the MIPS R10000 Processor.

The main observations from the results are : MDO optimized code has a larger spatial reuse of data both in the L1 and L2 cache. This increase in spatial reuse is due to the recursive allocation of simultaneously alive data for a particular cache size. This is observed from the L1 and L2 cache line reuse values. The L1 and L2 cache hit rates are consistently greater too, which indicates that the tile sizes evaluated by the tool were nearly optimal, since sub-optimal tile sizes will generate more self conflict cache misses.

	Initial	DOECU (I)	DOECU (II)
Avg memory access time	0.458275	0.293109	0.244632
L1 Cache Line Reuse	37.305497	72.854248	50.883783
L2 Cache Line Reuse	48.514644	253.450867	564.584270
L1 Data Cache Hit Rate	0.973894	0.986460	0.980726
L2 Data Cache Hit Rate	0.979804	0.996070	0.998232
L1–L2 bandwidth (MB/s)	115.431450	43.473854	49.821937
Memory bandwidth (MB/s)	10.130045	0.707163	0.315990
Actual Data transferred L1–L2 (in MB)	17.03	10.18	9.77
Actual Data transferred L2–Memory (in MB)	1.52	0.16	0.06

Table 2. Experimental Results for the voice coder algorithm using the MIPS R10000 Processor.

Since the spatial reuse of data is increased, the memory access time is reduced by an average factor 2 all the time. Similarly the bandwidth used between L1-L2 cache is reduced by a factor 0.7 to 2.5 and the bandwidth between L2 cache - main memory is reduced by factor 2-20. This indicates that though the initial algorithm had larger hit rates, the hardware was still performing many redundant data transfers between different levels of the memory hierarchy. These redundant transfers are removed by the modified data layout and heavily decrease the system bus loading. This has a large impact on the global system performance, since most (embedded) multimedia applications require to operate with peripheral devices connected using the off-chip bus. In addition also the system power consumption goes down.

	Initial	DOECU (I/II)
Avg memory access time	0.782636	0.289850
L1 Cache Line Reuse	9132.917055	13106.610419
L2 Cache Line Reuse	13.500000	24.228571
L1 Data Cache Hit Rate	0.999891	0.999924
L2 Data Cache Hit Rate	0.931034	0.960362
L1–L2 bandwidth (MB/s)	0.991855	0.299435
Memory bandwidth (MB/s)	0.311270	0.113689
Actual Data transferred L1–L2 (in MB)	0.62	0.22
Actual Data transferred L2–Memory (in MB)	0.20	0.08

Table 3. Experimental Results for the motion estimation algorithm using the MIPS R10000 Processor.

Since we generate complex addressing, we also perform address optimizations to remove the addressing overhead [5]. Our studies have shown that we are able to not only remove the complete overhead in addressing but gain by upto 20% in the final execution time measure on R10000 and PA-8000 processors, compared to the initial algorithm.

5 Conclusions

The main contributions of this paper are : (1) MDO is an effective approach at reducing the conflict cache misses to a large extent. We have presented a complete problem formulation and possible solutions. (2) This technique has been automated as part of a source-to-source precompiler for multimedia applications, called ACROPOLIS. (3) The results indicate a consistent gain in the data cache hit rate and a reduction in the memory access time and the memory bandwidth. Hence the system bus load and the energy consumption improve (reduce) significantly.

Acknowledgements : We thank Cedric Ghez and Miguel Miranda at IMEC for implementing the address transformations. Also we thank the ATOMIUM group for providing the C code parser and dumper.

References

1. P.Baglietto, M.Maresca and M.Migliardi, "Image processing on high-performance RISC systems", *Proc. of the IEEE*, vol. 84, no. 7, pp.917-929, july 1996.
2. D.C.Burger, J.R.Goodman and A.Kagi, "The declining effectiveness of dynamic caching for general purpose multiprocessor", *Technical Report*, University of Wisconsin, Madison, no. 1261,1995.
3. E.De Greef, "Storage size reduction for multimedia applications", *Doctoral Dissertation*, Dept. of EE, K.U.Leuven, January 1998.
4. F.Catthoor, S.Wuytack, E.De Greef, F.Balasa, L.Nachtergaele, A.Vandecappelle, "Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design", ISBN 0-7923-8288-9, Kluwer Acad. Publ., Boston, 1998.
5. S.Gupta, M.Miranda, F.Catthoor, R.Gupta, "Analysis of high-level address code transformations", In proc. of design automation and test in europe (DATE) conference, Paris, March 2000.
6. M.Kandemir, J.Ramanujam, A.Choudhary, "Improving cache locality by a combination of loop and data transformations", *IEEE trans. on computers*, vol. 48, no. 2, pp. 159-167, 1999.
7. C.Kulkarni, F.Catthoor, H.De Man, "Code transformations for low power caching in embedded multimedia processors,", *Intnl. Parallel Proc. Symp.(IPPS/SPDP)*, Orlando FL, pp.292-297,April 1998.
8. D.Kulkarni and M.Stumm, "Linear loop transformations in optimizing compilers for parallel machines", *The Australian computer journal*, pp.41-50, may 1995.
9. M.Lam, E.Rothberg and M.Wolf, "The cache performance and optimizations of blocked algorithms", *In Proc. 6th Intnl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp.63-74, Santa Clara, Ca., 1991.
10. C.L.Lawson and R.J.Hanson, "Solving least squares problems", *Classics in applied mathematics*, SIAM, Philadelphia, 1995.
11. N.Manjikian and T.Abdelrahman, "Array data layout for reduction of cache conflicts", *Intl. Conference on Parallel and Distributed Computing Systems*, 1995.
12. K.S.McKinley and O.Temam, "A quantitative analysis of loop nest locality", *Proc. of 8th Intnl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Boston, MA, October 1996.
13. G.L.Nemhauser, L.A.Wolsey, "Integer and Combinatorial Optimization", J.Wiley&Sons, New York, N.Y., 1988.
14. P.R.Panda, N.D.Dutt and A.Nicolau, " Memory data organization for improved cache performance in embedded processor applications", *In Proc. ISSS-96*, pp.90-95, La Jolla, Ca., Nov 1996.
15. P.R.Panda, H.Nakamura, N.D.Dutt and A.Nicolau, "Augmented loop tiling with data alignment for improved cache performance", *IEEE trans. on computers*, vol. 48, no. 2, pp. 142-149, 1999.
16. D.A.Patterson and J.L.Hennessy, "Computer architecture A quantitative approach", *Morgan Kaufmann Publishers Inc.*, San Francisco,1996.