Cache Conscious Data Layout Organization For Embedded Multimedia Applications

C.Kulkarni‡ C.Ghez M.Miranda F.Catthoor† H.De Man†

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
‡Also PhD student at the Katholieke Universiteit Leuven
†Professor at the Katholieke Universiteit Leuven

Abstract

Cache misses form a major bottleneck for real-time multimedia applications due to the off-chip accesses to the main memory. This results in both a major access bandwidth overhead (and related power consumption) as well as performance penalties. In this paper, we propose a new technique for organizing data in the main memory for data dominated multimedia applications so as to reduce majority of the conflict cache misses. The focus of this paper is on the formal and heuristic algorithms we use to steer the data layout decisions and the experimental results obtained using a prototype tool. Experiments on real-life demonstrators illustrate that we are able to reduce upto 82% of the conflict misses for applications that are already aggressively transformed at the source-level. At the same time, we also reduce the off-chip data accesses by upto 78% and combined with address optimizations we are able to reduce the execution time. Thus our approach is complimentary to the more conventional way of reducing misses by reorganizing the execution order.

1 Introduction and Related Work

The ever increasing gap between processor and memory speeds has motivated the design of 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 due to power consuming off-chip accesses to the main memory apart from increasing the average memory access time.

In this work, we are mainly targeting the embedded real-time multimedia processing (RMP) application domain which is an important growing market. Algorithms in there lend themselves to very good compile-time analysis and very aggressive data locality improving methods can be applied [6, 18]. 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 to largely fail in taking full advantage of the locality. According to [9], for StrongARM SA-110 about 43% of the total power consumption was due to the primary cache. Hence a maximal reduction of 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 [5, 10, 13, 14, 15] but still a significant amount of cache misses are present. Similarly storage order optimizations [5, 6] are very helpful in reducing the capacity misses. Thus mostly conflict cache misses related to the sub-optimal data layout remain. Array padding has been proposed earlier to reduce the latter [16, 18, 20]. These approaches are useful for reducing the (cross-) conflict misses to some extent. However existing approaches do not eliminate the majority of the conflict misses. Besides [3, 10, 18, 20], very little has been done to measure the impact of data layout(s) on the cache performance. Thus there is a need to investigate additional data layout organization techniques to reduce these remaining cache misses.

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

$$(Block\ Address)\ MOD\ (Nr\ of\ Sets\ in\ Cache) \quad (1)$$

Based on the number of lines in a set one can define direct mapped, n-way associative and fully associative cache [19]. 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. 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, termed as the main memory data layout organization in the sequel. The

formalized heuristic algorithm(s) to steer this forms our main contribution in this paper. They have been implemented in a prototype tool and its effectiveness 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 solution(s) in section 3. Experimental results on two large real-life applications and three smaller test-vehicles are presented in section 4. Some conclusions from this work are given in section 5.

2 Example Illustration

We now illustrate our data organization methodology on a compact but still representative real-life test vehicle namely a full search motion estimation kernel with four pixel accuracy [2]. This illustration is mostly qualitative, a more formal and quantitative approach is presented in section 3.

Typically, the memory allocation in traditional compiler/linkers is single contiguous and no cache parameters are taken in to account for this process. This is indicated as initial in figure 1. The second case in figure 1, indicated as improved initial, whose data layout is modified so as to incorporate the cache size. We have modified the base addresses of the arrays taking into account their life-times and cache size[1]. The third case in figure 1 is data layout optimized, where we first split the existing arrays and then merge them into groups based on the cache size and the line size, as explained in the succeeding sections. This data organization can be imposed on the linker by carefully rewriting the code (see [12]).

In the present example, we observe that, initially variables $Current[][]$ and $Previous[][]$ are mapped using the relation in equation 1. Thus whenever elements of $Current[][]$ and $Previous[][]$ are separated by a distance of the cache size, they will conflict with each other and cause additional cache misses. But for the data layout optimized case, variable $Current[][]$ and $Previous[][]$ can be mapped only to (mutually exclusive) cache locations 0 to 240 and 240 to 480 respectively. Thus we have eliminated the cross conflict misses altogether. A similar explanation holds for variables $V4x[][]$ and $V4y[][]$. In addition, since the number of partitions, due to tile sizes, of $Current[][]$ and $Previous[][]$ does not match those of $V4x[][]$ and $V4y[][]$, we have to let some of the locations, corresponding to the multiple of base address(es) of $V4x[][]$ and $V4y[][]$, remain empty so as to avoid the mapping of other variables on to those locations in cache. Thus we have some overhead in memory locations, in the present example 1%, but this is very reasonable and acceptable in our target domain as motivated in section 1. We impose the modified data layout by means of complex addressing of the form $((index)\%tilesize + \frac{index}{tilesize} \times cachesize + offset)$. Note that the complexity in addressing can be removed to a large extent using the address optimizations proposed in [7]. The effectiveness of that stage will be shown in section 4.

[1]By doing so we have eliminated some possibilities of cross conflict misses.

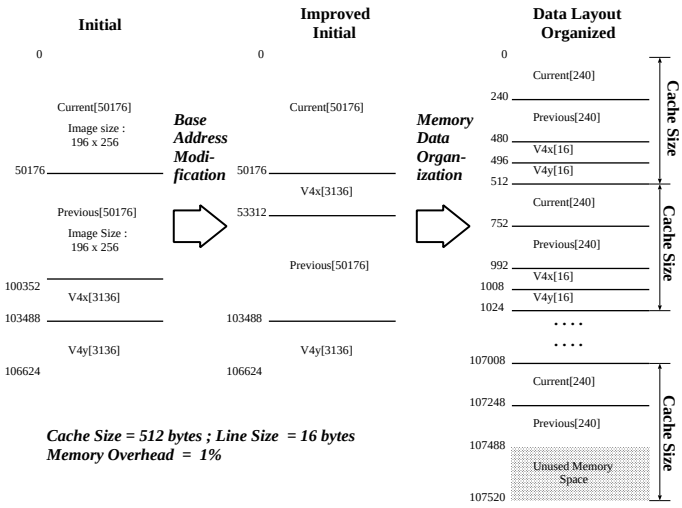

Figure 1. The initial and final data organizations for the motion estimation kernel.

We note that by performing the data layout organization as illustrated above, we are able to decrease the miss rates by upto 82% and reduce the write backs by upto 50%. More detailed results on different test-vehicles are available in section 4.

3 Data layout organization algorithm

In this section we present the algorithm to perform data layout organization that can be integrated in a compiler. First we will present a complete problem formulation involving the two stages namely the tile size evaluation and the array merging. 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 as a major step in the Acropolis source-to-source (C-to-C) precompiler.

3.1 Assumptions

We make two assumptions (be)for applying the data layout organization technique to any program: (1) Only the storage order of the program can be modified and all transformations which modify the execution order (like loop transformations [14, 15]) have already been applied and (2) Loop blocking [15] has been applied successfully and so is in-place data mapping [5] which ensure that the number of capacity misses is minimal. This is indeed the case, as observed in the experimental results (see section 4).

3.2 Basic terms

The following terms are used in this paper and they are interpreted (and defined) as below :

1. **Effective Size :** The total number of elements of an array accessed in a loop nest represents the effective size (ES_k) of the array in the particular loop nest.

2. **Tile Size :** The total number of contiguous lines of (main) memory allocated to an array in the cache is termed as tile size (x_k) of the particular array.

3. **Reuse Factor :** The ratio of total number of accesses to an array in a particular loop nest and the effective size of that array in the loop nest is termed as the reuse factor (RF_i) of that array in the particular loop nest.

3.3 Problem formulation

The general main memory data layout organization problem for conflict miss reduction 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. Before discussing the sub-problems, we first introduce the cache miss model which can be used to drive the optimization process.

3.3.1 Cache miss model

We now briefly present the cache miss model used to drive the optimizations in this work[2]. The conflict miss estimation has two main components namely the cross-conflict misses and the self-conflict misses. The main cost function used for this estimate and the algorithm used to estimate the number of misses are provided in figure 2. A more detailed explanation is omitted due to lack of space. It can be found in [12].

3.3.2 Tile size evaluation problem

The problem of tile size evaluation refers to the evaluation of the size of sub-array(s) for a given array (as shown in figure 1 and illustrated in section 2). 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 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.

[2]The cache miss model is presented here for completeness of the work and is not the main focus of this paper.

$$Cost\ Function = \sum_{offset=0}^{C-1} \frac{ES_j - ((ES_j + offset) - x_k)}{C} \times \frac{\#acc\ to\ j}{ES_j} + (ES_k - x_k) \times \frac{\#acc\ to\ k}{ES_k}$$

$$If\ (Effective\ size\ (of\ j) \leq C)\ then$$
$$\| \sum_{i=1}^{\#loops} \sum_{k=1}^{n} \sum_{j=1}^{\#c-vars} (Cost\ Function) \|$$
$$Else$$
$$\| \sum_{i=1}^{\#loops} \sum_{k=1}^{n} \sum_{j=1}^{\#c-vars} (\sum_{part=0}^{nparts} (Cost\ Function) + (Cost\ Function)_{ES_j = npart'}) \|$$

$$Where,$$
$$nparts = \frac{Effective\ size\ of\ j}{C}$$
$$npart' = (Effective\ size\ of\ j)\ MOD\ C$$
$$x_k \leq C$$

Figure 2. Pseudo-code for evaluating the total number of conflict misses in a given algorithm.

The second reason is that different loop nests have different number of arrays which are simultaneously alive.

$$\begin{aligned} L_1 &= x_1 + x_2 + x_3 + ... + x_n \leq C \\ &\cdots \\ L_m &= x_1^{(m-1)} + x_2^{(m-1)} + x_3^{(m-1)} + ... + x_n^{(m-1)} \leq C \end{aligned} \quad (2)$$

The above equations need to be solved so as to : (1) minimize the conflict misses obtained from algorithm in figure 2[3], (2) ensure that $0 < x_i \leq max(ES_i)$ and (3) ensure that $x_i\ mod\ L = 0$, where $i = 1 ... n$. The optimal solution to this problem comprises solving an ILP problem [17], which requires large CPU time for complex applications. Also, note that we can ensure an optimal solution only by imposing a strict equality to C in the above equations but for $n < m$[4], the strict equality does not guarantee a result and hence we use an inequality. To overcome this practical limitation, we have developed heuristics which provide good results in a reasonable CPU time, as demonstrated in section 4.

3.3.3 Array Merging/Clustering Problem :

We now further formulate the general problem using the loop weights for the heuristic approach. 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 :

$$L_{wk} = \sum_{i=1}^{n} ES_i \quad (3)$$

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

[3]In practice, we can also incorporate a model like cache miss equations [8] which should provide accurate cache miss information to be plugged into this framework.

[4]The total number of variables are less than the total number of loop nests.

this way because, we can have many tile sizes for each array[5] and there can be different number of arrays alive in different loop nests. In the example illustration in section 2 we have only one loop nest and hence we did not need this extension. Using the above considerations, we can identify loop nests which can potentially have more conflict misses (and assign corresponding weights) and focus on clustering arrays in the highest weighted loop nests (first).

3.4 Heuristic solution

We now discuss a pragmatic solution for the above problem. This solution makes use of a heuristic approach, which is less complex and faster from the point of view of implementation in a tool. The approach comprises the five steps explained below:

1. In the first step, we perform all the analysis. We evaluate the effective size of each array in each loop nest. Next, we also evaluate the number of accesses to every array in every loop nest.
2. In the second step, for every loop nest we evaluate the loop weights using the relation in equation 3.
3. Now, we visit the loop nest with highest loop weight. And we evaluate the individual array weights, where the array weight is the sum of reuse factors for the particular array in all the loop nests where it is alive times the effective size of the array in the considered loop nest.
4. In the fourth step, we obtain the tile size of all the arrays in the loop nest by proportionate allocation. The latter allocates larger tile sizes (in multiples of cache line sizes) to arrays with larger array weights and vice-versa. Once the tile size is obtained, we obtain the offset of the array in the cache through a global memory map used to keep track of all the array allocations.
5. We repeat the steps three and four for the loop nest with the next highest loop weight and so on, till all the arrays are covered. We perform code generation to fix the obtained data layout.

Note that in the above approach, we have solved both the tile size evaluation problem as well as the array merging problem in one step (step four). As mentioned earlier, this heuristic has been automated in a prototype C-to-C precompiler, which is based on the pseudo code shown in figure 3.

```
begin
  for everyLoopnest do
    for everyVariable do
      EffectiveSize = f(loopindex, arrayindex);
      EvaluateNrofAccesses(array);
    end
    LoopWeight = Σ_{∀variables} EffectiveSize;
  end
  for HighestWeightLoopnest do
    for everyVariable do
      if (Variable Not Allocated)
        ArrayWeight = Σ_{i=1}^{loops} ReuseFactor_i × ES;
        EvaluateTileSize(EffSize, ArrayWeight);
        Offset = UpdateGlobalMemMap(TileSize);
      end
    end
    Repeat until No variables are left unallocated;
  end
  for everyLoopnest do
    for everyVariable do
      ModifyCode(TileSize, Offset);
    end
  end
end
```

Figure 3. Pseudo-code for the heuristic data layout organization technique.

4 Experimental Results

In this section we present the experimental results obtained by applying the above discussed data layout organization tool on two large real-life demonstrators and three other (smaller) test-vehicles. The execution times were obtained after applying address optimizations [7] on the data layout organized code.

4.1 Metrics used

We have used three (main) metrics in this paper namely miss rate, memory bandwidth and power consumption. The memory bandwidth refers to the sum of number of cache misses and the number of write backs from the data cache to the off-chip main memory (on the system bus). The power consumption in the memories is evaluated using the power model similar to [13] and base energy values from [4]. Apart from the above three main metrics, in the experiments we have also included the total execution time of the program as well as the total number of instructions to provide a complete picture of the data layout organization technique.

4.2 Experimental setup

The experimental setup comprises two parts: (1) The prototype C-to-C precompiler is coupled to the cache simulator and the processor (board). We thus obtain the transformed C code from the prototype data layout transformation tool and compile this transformed C code with native compiler of the

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

simulator or the processor. (2) We have used the SimpleScalar simulator tool set [21] in this work for simulating cache performance for varying cache sizes. We have also used real processors for observing performance. The processors used are PA-RISC 8000 and MIPS R10000.

4.3 Results and discussion

We present a discussion of the experimental results obtained in this section. We will discuss the impact of data layout organization technique on the data cache miss rate as well as on the number of off-chip data accesses. Followed by this we will briefly discuss the influence of the data layout organization technique on the number of instructions and the number of cycles.

Table 1 shows the miss rate and the number of off-chip accesses for the cavity detection and the QSDPCM algorithms. We observe that the miss rate is consistently reduced by 65-70% on the average and for some cases it is reduced by upto 82%. This implies that we are able to remove a large majority of the conflict misses by the data layout organization technique. Note that our data layout organized code on direct mapped cache is able to outperform the initial code even on a 2-way associative cache[6] in terms of miss rate. The reduction in the number of off-chip accesses also follows a similar pattern as observed in table 1, which also means that the data layout technique is able to reduce the write backs apart from the conflict misses. Table 1 also shows that the miss rate and the number of off-chip accesses for the SOR, the motion estimation and the 2d convolution algorithms. Here too we observe that the miss rate reduces consistently and so are the off-chip accesses, confirming again the large impact of the data layout organization technique. Also, we note that the initial cavity detection algorithm on a direct mapped cache consumes 146mW, 177mW, 312mW for 512 byte, 1Kbyte and 2Kbyte cache sizes respectively [4]. The corresponding data layout organized algorithms requires 79mW, 151mW and 286mW respectively. This illustrates the reduction in power consumption due to our technique.

Our initial goal was to achieve the performance (in terms of miss rate) of a fully associative cache using a direct mapped cache. This study was intended to show that by increasing the control complexity in the compiler, we can reach a performance close to the complex hardware control embedded in a fully associative cache (which is much more expensive in terms of power and area). We have achieved this goal because we are within 18% of the theoretical limit, as seen in table 1. Note that the data layout organized case for a direct mapped cache (6.39% miss rate) performs better than the initial two way associative case (9.92% miss rate), which illustrates that our technique is able to outperform a 2-way associative cache without paying the hardware overhead in power and area.

[6] A 2-way associative cache also consumes more power than a direct mapped since the number of tag bits increase (more bit lines and related switching activity) and the increase in number of comparators.

	Miss Rate			Mem BW		
	512	1K	2K	512	1K	2K
Cavity Detection						
Initial - DM	35.5	9.10	5.72	43.88	12.84	8.32
DL Orgd - DM	6.39	3.90	2.30	9.26	5.65	3.48
Initial - FA	1.95	0.25	0.25	2.93	0.49	0.44
Initial - 2Way	9.92	5.44	2.37	13.5	7.84	3.54
DL Orgd - 2Way	3.2	1.50	1.30	4.41	2.27	1.98
Initial - FA	1.95	0.25	0.25	2.93	0.49	0.44
QSDPCM						
Initial - DM	13.78	9.85	7.14	15.74	11.25	8.21
DL Orgd - DM	9.39	5.18	2.98	11.40	6.20	3.65
Initial - FA	2.3	2.12	1.58	2.93	2.63	2.12
SOR						
Initial - DM	86.40	78.21	45.50	94.90	86.63	49.80
DL Orgd - DM	51.37	46.09	34.79	58.70	53.40	38.90
Initial - FA	37.22	29.37	29.37	40.87	31.40	25.42
Motion Estimation						
Initial - DM	60.63	60.20	47.53	62.38	62.32	49.41
DL Orgd - DM	48.60	37.30	26.30	49.44	38.10	27.10
Initial - FA	1.16	1.12	1.11	1.84	1.21	1.11
2D Convolution						
Initial - DM	13.69	8.77	5.60	14.94	9.80	6.51
DL Orgd - DM	9.40	3.87	3.10	10.50	4.60	3.90
Initial - FA	1.59	1.59	1.59	2.35	2.23	1.76

Table 1. Cache miss rates and the memory bandwidths for different algorithms and different associativities of cache. Cache size are in bytes, DM stands for direct mapped and FA stands for fully associative.

We have earlier shown the impact on the data cache miss rates due to the data layout organization technique. The actual implementation of this technique involves modification of address values, which adds to the number of instructions. Table 2 shows the number of cycles as well as the number of instructions for cavity detection algorithm on SimpleScalar machine simulated (sim-outorder) with a 512 byte direct mapped data cache, 2Kbyte 4-way instruction cache, single cycle access on a cache hit, a penalty of 18 cycles on a miss and no level two caches. Note that the overhead in instructions is approximately 21%. We observe that we are able to gain in the total cycles due to the reduction in conflict misses for the data cache by 82% even though there is an increase in the total number of instructions. It is worth noting that the Simplescalar architecture has dedicated instructions for integer division, which is a single cycle access. This though is not true for any of the existing (commercial embedded) processors. We perform systematic address code transformations [7] on this code. We now use existing processors to illustrate that we can remove all the overhead in cycles due to complex addressing and reduce the number of cycles even further due to reduced cache misses. The address transformations especially target the integer division and modulo's introduced by the data layout technique. Table 3 shows that indeed we are

able to remove majority of the overhead for both the cavity detection and motion estimation algorithms for two different platforms. However note that the impact of these transformations is highly platform and application specific and hence there is a need for tool support for these steps to explore different alternatives. Additional results on this technique are available in [11].

	Initial code	Data layout organized
# cycles	290M	230M
# instructions	323M	391M

Table 2. Simulated Number of cycles and number of instructions for cavity detection algorithm.

	Initial	Global Trf + Adopt	Global Trf + DL + Adopt
Cavity Detection			
Exec Time (PA-8000)	0.77s	0.23	0.26s
Exec Time (MIPSR10k)	2.90s	1.19	0.97s
Motion Estimation			
Exec Time (PA-8000)	-	80ms	50ms
Exec Time (MIPSR10k)	-	218ms	211ms

Table 3. Execution time for cavity detection and motion estimation algorithms.

5 Conclusion

In summary we observe the following from the above results: (1) The data layout organization technique is able to reduce conflict misses significantly for all the drivers for different cache sizes. For larger real-life applications like cavity detection and QSDPCM we are able to achieve upto 82% reduction in the conflict misses, and (2) For embedded systems which are bandwidth constrained, this technique is able to reduce the off-chip accesses to a large extent. This is a significant design issue which makes this technique much more useful than existing techniques focusing solely on performance.

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] V.Bhaskaran and K. Konstantinides, "Image and Video Compression Standards : Algorithms and Architectures", Kluwer Acad. Publ., Boston, 1995.

[3] P.Clauss, B.Meister, "Automatic memory layout transformation to optimize spatial locality in parametrized loop nests", *4th Annual Workshop on Interaction between Compilers and Comp. Arch. (INTERACT-4)*, Toulouse, France, Jan 2000.

[4] J.Covino, J.Connor, D.Evans, A.Roberts, M.Robillard, J.Sousa, L.Ternullo, "A 2ns zero wait state, 32KB semi-associative L1 cache", *IEEE Intl Solid State Circuits Conference (ISSCC-96)*, pp. 154-155, 1996.

[5] E.De Greef, "Storage size reduction for multimedia applications", *Doctoral Dissertation*, Dept. of EE, K.U.Leuven, January 1998.

[6] 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.

[7] C.Ghez, M.Miranda, A.Vandecappelle, F.Catthoor, D.Verkest, "Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm", *Proc. IEEE Wsh. on Signal Processing Systems (SIPS)*, Lafayette LA, IEEE Press, Oct. 2000.

[8] S.Ghosh, M.Martonosi, S.Malik, "Cache Miss Equations : A Compiler Framework for Analyzing and Tuning Memory Behaviour", *ACM transactions on Programming Languages and Systems*, vol. 21, No. 4, pp. 702-746, July 1999.

[9] N.Jouppi, et al. , "A 300-MHz 115-W 32-b bipolar ECL microprocessor", In IEEE Journal of solid-state circuits, pp. 1152-1165, Nov 1993.

[10] 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.

[11] C.Kulkarni, F.Catthoor, H.De Man, "Advanced data layout optimization for multimedia applications", *Lecture notes in computer science (PDIVM/IPDPS 2000)*, vol. 1800, pp. 186-193, May 2000.

[12] C.Kulkarni, F.Catthoor, H.De Man, "Cache optimization for multimedia applications", *Doctoral Dissertation*, Dept. of EE, K.U.Leuven, February 2001.

[13] 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.

[14] D.Kulkarni, M.Stumm, "Linear loop transformations in optimizing compilers for parallel machines", *The Australian computer journal*, pp.41-50, may 1995.

[15] M.Lam, E.Rothberg, 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.

[16] N.Manjikian and T.Abdelrahman, "Array data layout for reduction of cache conflicts", *Intl. Conference on Parallel and Distributed Computing Systems*, 1995.

[17] G.L.Nemhauser, L.A.Wolsey, "Integer and Combinatorial Optimization", J.Wiley&Sons, New York, N.Y., 1988.

[18] P.R.Panda, N.Dutt, A.Nicolau, "Memory issues in embedded systems-on-chip", *Kluwer Academic Publishers*, Boston, MA, 1999.

[19] D.A.Patterson, J.L.Hennessy, "Computer architecture: A quantitative approach", *Morgan Kaufmann Publishers Inc.*, San Francisco,1996.

[20] G.Rivera, C.Tseng, "Compiler optimizations for eliminating cache conflict misses", *Technical Report CS-TR-3819*, Dept of Computer Science, University of Maryland, July 1997.

[21] D.Burger, T.Austin, "The Simplescalar Toolset", Version 2.0, online document available at http://ww.cs.wisc.edu/ mscalar/simplescalar.html.