Memory Requirement Optimization with Loop Fusion and Loop Shifting

Qubo Hu[1] Martin Palkovic[2] Per Gunnar Kjeldsberg[1]
[1] Norwegian University of Science and Technology, Trondheim, Norway
[2] IMEC, Leuven, Belgium
{qubo.hu, per.gunnar.kjeldsberg}@iet.ntnu.no, palkovic@imec.be

Abstract

Loop fusion and loop shifting are well recognized loop transformations for memory requirement reduction. State-of-the-art optimizations with loop fusion and shifting are based on heuristics without any evaluation of the resulting effects during each optimization step. Thus we cannot guarantee that each step results in a reduced overall memory requirement. On the other hand, most memory requirement estimations at system level are inefficient and slow. Also the estimation is not started until the optimization is done. Having to iterate between optimization and estimation is very time consuming. In this paper, we present a storage requirement optimization method which combines the optimization and estimation processes with the goal to have continuous estimates during the optimization and hence to achieve lower memory requirements.

1. Introduction

In today's embedded systems, the memory hierarchy is rapidly becoming a major bottleneck in terms of power, performance and area. This is especially the case for embedded multimedia and communication applications, which are characterized by deep loop nests and multi-dimensional arrays. It has been shown that between 50% and 80% of the global digital power in an embedded multimedia system is consumed by data transfer and storage (as opposed to the computation in processor(s)) [7]. As a result, much effort has been devoted to memory hierarchy optimization, especially at system level due to the steadily increasing size and complexity of integrated circuits. For example, this has led to the so-called Data Transfer and Storage Exploration (DTSE) methodology [4, 3] and appropriate tool support technique developed at IMEC.

An important early system level technique, the loop transformation technique, is aiming at improving the data access regularity and locality and removing the system-level buffers of the application codes. Hence it reduces the overall memory size requirement and the access frequency to big and slow memories. This is vital to area, power consumption, and performance. Improved data access regularity and locality shorten the lifetimes of data elements and increases the memory location reuse ratio since memory locations can be reused for data elements with non-overlapping life-times. This in turn reduces the memory size requirement. However, current data locality improvement algorithms for memory optimization (e.g., [18, 17, 6]) are based on heuristics without estimating their effect. It can lead to final sub-optimal solutions since optimizing locality may in some cases actually deteriorate memory usage. On the other hand, memory estimation (e.g., [19, 9, 10]) has also been studied but they have so far not been used to guide the optimization.

In this paper, we propose a more efficient memory optimization method when improving data access locality with loop fusion and shifting. This is done by interactively estimating the memory size requirement during the optimization procedure, in order to take into account the mutual effects of all data locality improvements and achieve the global memory size optimization. Loop fusion is a code transformation that takes related loop nests that have the same iteration space traversal and combines their bodies into a single loop, enhancing data locality and reducing the memory requirement. Loop shifting is introduced when fusion is not legal because data dependencies in the fused loop exist for which statements of one loop iteration depend on results from statements of the following loop iterations.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 details our memory requirement optimization and estimation methodology. Subsection 3.1 introduces the dependency graph used and subsection 3.2 describes our algorithm for memory requirement optimization and estimation. The algorithm is illustrated using a one-dimensional example code. Section 4 holds further discussion while conclusions are finally drawn in section 5.

2. Previous Work

2.1. Memory minimization and data locality improvement

Loop transformations have been studied quite extensively for optimization of parallelism and performance. We will, however, only concentrate on loop fusion and loop shifting for memory minimization and locality improvement in this paper. Manjikian *et al.* [14] only considers loop shifting as soon as loop fusion is illegal and therefore it can only reach partial locality improvement. Fraboulet *et al.* [6] considers minimizing the memory accesses in loop nests by data temporal locality optimization. They present a polynomial algorithm for memory access optimization in a single loop and a heuristic based on this algorithm for the multidimensional case and also the maximal dependency distance minimization for a loop nest. However, it does not guarantee optimal memory access minimization since only the last accesses to the given array is optimized when several statements read this array. Song *et al* [17] use a greedy heuristic to incrementally fuse loop nests when the fusion results in less register spilling and cache misses. Otherwise, the fusion will not be done.

As opposed to these works which mostly look at uniform dependencies (or even some exceptional non-uniform dependencies), Verdoolaege *et al.* [18] proposes a greedy algorithm for incremental loop fusion. It works with all general types of dependencies including uniform, non-uniform and cyclic dependencies. Initially all iteration domains are represented as nodes (see Section 3.1 for an explanation of iteration domain). At each step, only the two nodes for which the dependency between them involves the highest number of data elements are considered for fusion. This process continues until there is just one node left. The technique achieves a reasonable result within a reasonable time complexity. In addition to this, much work focus on optimizing the memory system at other design stages. For example, [13] improves data locality using affine partitioning. Greef [8] optimizes memory usage by data layout optimization through both inter-array and intra-array in-place techniques. In fact, [18] and the method proposed in this paper are just an enabling step for the actual memory optimization performed in [8].

As shown, heuristics are generally used for loop fusion and shifting. In fact, [5] has proven that loop shifting for array contraction is NP-complete even for acyclic dependency graphs with uniform dependencies.

2.2. Memory size estimation

Traditionally memory requirement optimization was scalar-based. Balasa *et al.* [2] presents one of the first papers dealing with system level memory size estimation for applications with arrays. With his approach, arrays are partitioned into non-overlapped basic sets. A data-flow graph is built where the nodes represent basic sets weighted with the number of scalars in it. The arcs represent dependencies between basic sets weighted with the number of dependencies. Each basic set can now be treated as a single unit. The total storage requirement is found through a greedy traversal of the data-flow graph. This method takes into account that signals in different basic sets with non-overlapping lifetime can share the same memory location. It does not take into account the execution ordering, which gives rise to overestimation.

Compared to this, the work done by Zhao *et al.* [19] and Grun *et al.* [9] assumes a fixed executing ordering for the application code. [19] proposes a method that measures the number of simultaneously alive variables in each iteration. This counting of live array elements is done by set operations (union, intersection) which are the whole or parts of the iteration domains. In [9], the data dependency relations between the array references in the code are used to find bounds on the number of array elements produced or consumed by each assignment. Then, a memory trace as a function of time is found. The peak memory trace contained within the bounding rectangles yield the total memory requirement. If the difference of boundaries for the critical rectangle is too large, the corresponding loop is split and the estimation is rerun in order to improve the estimation accuracy. In a worst case, a full loop unrolling is required to achieve a satisfactory estimate, which is unaffordable.

Finally, the work done by Kjeldsberg *et al.* [10, 11, 12] is reviewed. The memory size estimation is done when the execution ordering is only partially fixed. It can suggest the most beneficial execution ordering with regard to the estimated memory requirement. With this approach, the size of individual dependencies is estimated. The lifetime of all the individual dependencies are analyzed and dependencies which are simultaneously alive are identified. The set of simultaneously alive dependencies requiring the largest memory requirement decides the total memory requirement. When the execution ordering is not fully fixed, the estimate results in upper and lower bounds of the memory requirement.

In summary, the approaches for system-level memory minimization rely on heuristics that can not always guarantee an optimal global solution. Meanwhile, all previous work of the high-level memory size estimation has been processed either by itself or separately from an optimization procedure. This is not effective to achieve global optimization result. It is necessary to combine the two steps and do interactive estimation and optimization in order to achieve global memory optimization.

3. Combined Memory Requirement Optimization and Estimation

This memory optimization method is suited for applications in which contradictive locality improvements may occur. This means that a possible improvement of locality for one set of data can have negative effects on the locality for other sets of data. In this case, the previous methods for memory requirement optimization will certainly not guarantee optimal results. Our method estimates the optimization effort based on a global view of the memory consumption and performs improvements if the peak memory requirement can possible be reduced by a shifting of the corresponding iteration domains. At this moment, we consider single assignment code where every array value can only be written once but read several times. We look at acyclic dependencies with perfect loop nests, which means all statements are in the inner most level.

3.1. Program model

The high level application code is typically described as a number of statements accessing multi-dimensional arrays. The statements are enclosed in loop nests and possibly guarded by conditions with all the loop bounds and conditions being affine expressions of loop iterators and structural parameters. The loop iterators give rise to a multi-dimensional iteration space, where each iteration node corresponds to one iteration of the loop nest. Each statement in the loop nest has an Iteration Domain (ID), which contains the set of iteration nodes for which the statement is executed. The IDs can be represented by polytopes (convex sets bounded by hyperplanes). In this representation, each dimension corresponds to a loop iterator. For the example code shown in Fig. 1, the iteration domains A and D are two sets of iteration nodes for which each statement is executed. As the execution of iteration domain D depends on the execution of iteration domain A, we say that D is a dependent iteration domain depending on A and there are dependencies between them. The (sub-)set of iteration nodes of iteration domain A which iteration domain D depends on is called Dependency Part (DP).

The dependency is described as a distance vector (DV), which measures the dependence distance in each dimension and is calculated as the difference between the iterator values involved in the dependency. All information about iteration domains and dependencies can be described in a Dependency Graph (DG) with the representation of $G = (V, E, DP)$. V symbolizes the set of all iteration domain polytopes $< (P_{S_i}) \mid 1 \leq i \leq n >$ and n is the number of iteration domains. E symbolizes the set of distance vectors $< (DV_{S_i S_j}) \mid 1 \leq i \leq n, 1 \leq j \leq n, i \neq j, S_j$ depends on $S_i >$ for all dependencies. DP represents the set of dependency parts $< (DP_{S_i S_j}) \mid 1 \leq i \leq n, 1 \leq j \leq n, i \neq j, S_j$ depends on $S_i >$. For the one-dimensional example with the source code shown in Fig. 1.a, the iteration domain A and D are represented as polytope $P_A = \{(i) | 0 \leq i \leq 5\}$ and $P_D = \{(i) | 1 \leq i \leq 4\}$ respectively. The dependency part for dependencies between A and D is [1, 2, 3, 4] (simply written as [1,4]). The distance vector between A and D is expressed as $DV_{AD} = [0]$, which is the difference of iterator values in the i-dimension for each dependency between the depending iteration domain A and the dependent iteration domain D. For more details of the dependence graph, please refer to [1]. There are normally a number of dependencies between a pair of iteration domains, one for each iteration node in the depending domain. From now on, we refer to all dependencies between a pair of iteration domains as a pair dependency.

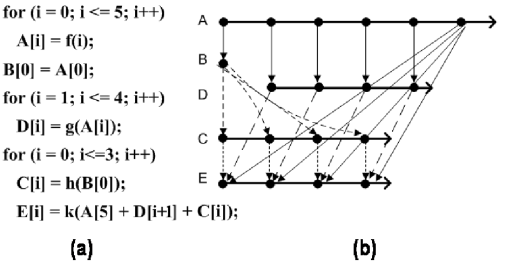

Figure 1. Program example with dependencies

3.2. Algorithm description

Before describing our algorithm, some concepts need to be clarified. The Memory Requirement (MR) for one pair dependency equals the number of memory locations needed to save the data elements of the dependency. Since memory locations can be reused by data elements with non-overlapping lifetimes, the MR equals the number of iteration nodes visited in the DP before the first dependent iteration node is visited. It can be calculated based on DP and DV. The lifetime of one pair dependency is defined as a space range of iteration nodes, which is the union of iteration nodes from the dependency part and the dependent iteration domain. The total memory requirement of an application is caused by the Peak Memory Requirement (PMR) encountered during its execution. This is the largest sum of memory requirements for a set of pair dependencies whose lifetimes are overlapping. Such a set of pair dependencies are called simultaneously alive dependencies (the concept has been similarly used in [10, 16]).

Memory requirement estimation and optimization are integrated in our method and are performed interactively.

1 Extract necessary information for the dependency graph (***DG***)
2 Fuse dependent Iteration Domains. Shifting is only done when fusion is not legal
3-1 Calculate ***MR*** and lifetimes for all pair dependencies
3-2 Calculate all sets of simultaneously alive dependencies and their combined ***MR***
Find the set with the largest combined ***MR***, which is the *current**PMR**set*
4-1 Find all Shiftable Iteration Domains(***SID***) and their lifetimes with **ASAP/ALAP**
Mark all shiftable iteration domains(*all_**SID***) unshifted
*Unshifted**SID**set* = *all_**SID***
LIST = *all_**SID*** ∩ *current**PMR**set*
5 *FurtherImprovement* = True
While *(FurtherImrprovement)* and *(Unshifted**SID**set* ≠ empty*)*
5-1 Select one ***SID*** from the ***LIST***
5-2 If (shifting the ***SID*** improves ***PMR***) # *Evaluate possible shifting*
mark the ***SID*** shifted
*Unshifted**SID**set* = *Unshifted**SID**set* – ***SID***
If (*current**PMR**set* does not result in ***PMR*** any more after shifting)
5-2-1. Re-find the set with the largest combined ***MR***,
which will be the *current**PMR**set*
5-2-2. ***LIST*** = *unshifted**SID**set* ∩ *current**PRM**set*
else ***LIST*** = ***LIST - SID***
else ***LIST*** = ***LIST*** – ***SID***
5-3 If (***LIST*** == empty) *FurtherImprovement* = False
6 The current ***PMR*** is the result of optimized total memory requirement
7 Transform to the output code based on ***DG***

Figure 2. Outline of our algorithm for Memory Optimization together with iterative estimation

Fig. 2 gives an outline of our algorithm. We will now give some more details regarding the different steps. In step 1 we extract all necessary information from the application source code and build up the dependency graph. Fig. 3 shows the dependency graph for the example code of Fig. 1.

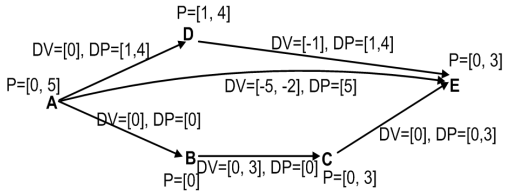

Figure 3. Dependency graph

In step 2 we fuse all dependent iteration domains into a common iteration space. If fusion is not legal, iteration domains are shifted to make it legal. In this step, we do not try to shift dependencies as close as possible. Shifting is only performed when fusion is not legal as explained above. As an example, iteration domains D is fused directly with A while fusing E directly with A and D is not legal. Iteration domain E is hence shifted during fusion and the corresponding polytope P_E is changed from $P_E = \{(i)|0 \leq i \leq 3\}$ to $P_E = \{(i)|5 \leq i \leq 8\}$. Related dependencies hence need to be updated. The dependency graph and the dependency diagram at the end of this step are shown in Fig. 4.

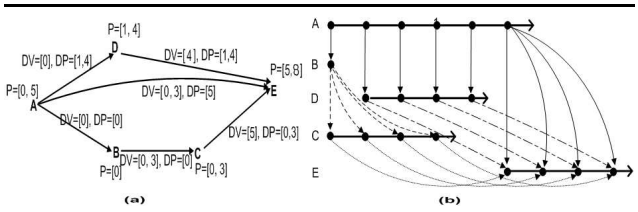

Figure 4. Fusing program with corresponding dependencies

In step 3-1, the memory requirement and lifetime of each pair dependency are calculated. In our example, the memory requirements (MR) for each pair dependency are $MR_{AB} = 1$, $MR_{AD} = 1$, $MR_{BC} = 1$, $MR_{DE} = 4$, $MR_{CE} = 4$ and $MR_{AE} = 1$ respectively. The lifetimes (LF) of them are, *e.g.,* $LF_{AB} = [0]$, $LF_{BC} = [0, 3]$, and so on. Then, in step 3-2, all sets of simultaneously alive dependencies are found by intersecting the lifetimes of all pair dependencies. The memory requirement of each set is calculated by summing up the memory requirement for all pair dependencies in the set. The set which consumes the peak memory requirement among all sets is chosen as *currentPMRset*. In this case, the set containing dependencies $\{DP_{BC}, DP_{CE}, DP_{DE}, DP_{AD}\}$ causes the peak memory requirement. The memory requirement of it is 10 memory elements. Actually, the memory requirement is 9, not 10. This is because dependency DP_{DE} can reuse the same memory locations required for DP_{AD}. This kind of overestimation will be further discussed in section 4. So the actual peak memory requirement is 9.

The iteration domains are categorized as fixed iteration domain or unfixed iteration domain. The fixed iteration domains can not be shifted. An unfixed iteration domain is a shiftable iteration domain (SID) if it can be placed at different positions in the iteration space while still keeping all dependencies legal at the end of fusion and shifting. Otherwise, it becomes a fixed iteration domain. The freedom of a shiftable iteration domain indicates the range in the iteration space to where it can be legally shifted. The iteration domain that does not depend on any other iteration domains and whose first iteration node has the smallest value is defined as the fixed input iteration domain. The iteration domain that no other iteration domains depend on and whose last iteration node has the largest value is defined as the fixed output iteration domain. If there are more than one in-

put iteration domain having identical smallest first iteration node, they are all fixed input iteration domains. The same case goes for fixed output iteration domains. In the example, iteration domains A and E are fixed input and output iteration domains respectively. B, C and D are unfixed iteration domains.

In step 4-1, we do both As Soon As Possible (ASAP) and As Late As Possible (ALAP) placements for all unfixed iteration domains in order to find the shiftable iteration domains and their freedom. When doing ASAP placement, we try to shift all unfixed iteration domains as close as possible to the fixed input iteration domains, following the data flow procedure. Thus we get the earliest possible start positions of these unfixed iteration domains. For ALAP placement, all the unfixed iteration domains are shifted as close as possible to the fixed output iteration domains, following the opposite data flow procedure. This results in the latest possible start position of these iteration domains. The earliest and latest start positions defines the freedom to where the unfixed iteration domains are allowed to be shifted. These ASAP and ALAP scheduling have similarly been used in [15] but they are only used for scalar-based memory optimization. In the example, B, C and D are shiftable iteration domains and their freedom are [0, 4], [0, 4] and [1, 5] respectively. All of them are initialized as unshifted. The variable *UnshiftedSIDset* contains all unshifted iteration domains $\{B, C, D\}$. In step 4-2 we find the shiftable iteration domains $\{B, C, D\}$ that the *currentPMRset* contains and assign them to *LIST*.

In step 5 the core optimization and estimation are performed interactively. *FurtherImprovement* is initialized to be True. The while loop is executed as long as both conditions are true. The shiftable iteration domain having the largest absolute weight value is selected from the *LIST*. The weight is calculated based on the formula below. This formula is used to decide which shiftable iteration domain should be considered for shifting first and also in which direction it should be shifted. A positive weight value indicates that the domain has potentially bigger shifting benefit in the direction towards the fixed output iteration domain while a negative weight value indicates bigger shifting benefit in the opposite direction. This is because the positive weight value indicates stronger dependency relations between *N* and the iteration domains that depends on *N* and may give rise to a larger memory requirement reduction possibility when shifted in this direction. The same reasoning goes for negative weight values.

$$\begin{aligned} weight = & \quad \Sigma DPS_{Nj} * MR_{Nj} * |DV_{Nj} + 1| \\ & - \quad \Sigma DPS_{iN} * MR_{iN} * |DV_{iN} + 1| \\ & \quad (1 \leq i \leq n, 1 \leq j \leq n, i \neq j) \end{aligned}$$

In this formula, *j* indicates the iteration domain that depends on iteration domain *N*. *i* indicates the iteration domain that *N* depends on. DPS_{Nj} and DPS_{iN} mean the size of the dependence part for the dependencies involved. MR_{Nj} and MR_{iN} mean the memory requirement for the pair dependencies involved. $\Sigma DPS_{Nj} * MR_{Nj} * |DV_{Nj} + 1|$ represents the sum of weights for all the dependencies that depends on the iteration domain *N*. $\Sigma DPS_{iN} * MR_{iN} * |DV_{iN} + 1|$ represents the sum of weights for all the dependencies that the iteration domain *N* depends on. That means, for any shiftable iteration domain *N*, we calculate the weights individually in two directions: backwards to the fixed input iteration domain and forwards to the fixed output iteration domain. Three parameters are involved in the calculation of weight, the size of dependency part (DPS), the memory requirement (MR) and the distance vector (DV) for every pair dependency involved. These three parameters is chosen with the following reasoning:

- With a bigger dependency part (DPS), a shifting of an iteration domain can result in bigger memory requirement change.
- If the current memory requirement (MR) is big, it has a bigger potential for optimization.
- When the distance vector (DV) is bigger, the shifting effect is potentially bigger

All three parameters may affect the results of a shifting. Besides, there are also mutual relationships between them. For example, the dependency having bigger distance vector and bigger dependency part requires bigger memory requirement. The memory requirement will decrease as soon as the associated iteration domain is shifted in the corresponding direction. The reasoning behind the multiplication with $|DV + 1|$ instead of just DV is that it is necessary to take into account dependencies having zero distance vector. Simply ignoring them for dependencies having different dependency parts will give a loss in accuracy. A more detailed investigation of the consequences of alternative weight calculations is a part of our future work.

The weight for shiftable iteration domains B, C and D are 0, 95 and 76 respectively. C is consequently chosen for shifting towards the fixed output iteration domain. We find that the peak memory requirement is reduced from 9 to 7 when *SID* C is shifted from $P_C = \{(i)|0 \leq i \leq 3\}$ to $P_C = \{(i)|5 \leq i \leq 8\}$. The related dependencies need to be updated as a result of the shifting. *SID* C is hence marked shifted and is removed from *UnshiftedSIDset* in step 5-2. *UnshiftedSIDset* now just contains {B, D}. Since the *currentPMRset* still contributes to the peak memory requirement after the shifting of C, steps 5-2-1 and 5-2-2 are not executed and *SID* C is removed from *LIST*. Neither *LIST* nor *UnshiftedSIDset* are empty, so the while loop continues. In the second iteration, *SID* D is chosen but it does

not improve the peak memory requirement with any possible shifting. After this iteration, *LIST* = {B} and *UnshiftedSIDset* = {B, D}. During the third iteration of the while loop, *SID* B is tried but it does not improve the peak memory requirement either. Then *LIST* = {} and *UnshiftedSIDset* = {B, D}. As *LIST* == {}, the variable *FurtherImprovement* = False and the while loop terminates.

The final peak memory requirement of 7 is the optimized total memory requirement. The final dependency diagram and corresponding code are shown in Fig. 5.

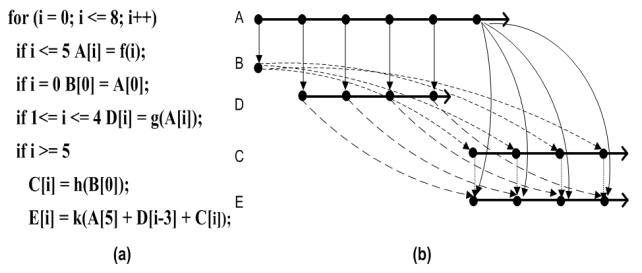

Figure 5. Fused program with corresponding dependencies

4. Discussion

We will now discuss some future extensions that can reduce the time complexity and improve the optimization accuracy.

As indicated in the previous Section, dependencies DP_{AD} and DP_{DE} can reuse a memory location. This is due to the fact that when D depends on A and E depends on D, the memory location used for DP_{AD} can immediately be reused for DP_{DE}. This is the case when both DP_{AD} and DP_{DE} are uniform dependencies. It would be complex in the case any of DP_{AD} and DP_{DE} are not uniform. But it is still possible to remove such overestimates.

As described , the simple way to calculate the memory requirement for a set of simultaneously alive dependencies is to sum up the memory requirement of all pair dependencies in the set. With such a simple approach, overestimation may occur when there is a case where two or more pair dependencies depends on the same iteration domain and their dependency parts are overlapping. If all these pair dependencies are uniform dependencies, overestimation can be removed by just taking the union of the sets of iteration nodes visited in the dependency part before the first dependent iteration nodes of the pair dependencies are visited.

As it is an NP-problem to decide the order in which shiftable iteration domains should be considered for shifting and to where they should be shifted, a greedy algorithm is used to select one iteration domain at a time. This algorithm will not detect a case in which no single shift decreases the peak memory requirement. In order to reduce the computation costs we currently only consider a limited number of places when trying to shift iteration domains around. As a result, our method can not always guarantee an optimal solution. It is expected to achieve good optimization result, based on the global view, by interactively estimating the possible memory optimization effects. Besides, our method could possibly improve optimizations of other approaches, *i.e.*, [18, 17, 10] by using their results as input. The methodology can work for applications with non-uniform dependencies as illustrated.

For future work, it would also be useful to remove the limitations of the code format requirement to a more general case. As a larger benefit can be gained through memory requirement optimization of multi-layer memory architectures, we are currently looking at these issues based on this work.

5. Conclusion

This paper presents a technique for memory requirement optimization with loop fusion and loop shifting, which will result in the globally reduced memory requirement. Compared to previous work, the technique combines the memory requirement estimation interactively with the optimization efforts in order to achieve a globally optimized result.

References

[1] D. F. Bacon, S. L. Grahan, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM computing surveys*, 26(4):245–420, Dec. 1994.

[2] F. Balasa, F. Catthoor, and H. De Man. Background memory area estimation for multi-dimensional signal processing systems. *IEEE Trans. on VLSI Systems*, 3(2):157–172, June 1995.

[3] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. V. Achteren, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Acad. Publ., Boston, USA, 2002. ISBN 0-7923-7689-7.

[4] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Acad. Publ., Boston, USA, 1998. ISBN 0-7923-8288-9.

[5] A. Darte and G. Huard. New results on array contraction. In *IEEE Conference on Application-Specific Systems, Architectures, and Processors (ASAP'02)*, pages 359–371, 2002.

[6] A. Fraboulet, G. Huard, and A. Mignotte. Loop alignment for memory accesses optimization. In *in Twelfth International Symposium on System Synthesis Proceedings*

(ISSS99), IEEE Computer Society Press, pages 71–77, Nov. 1999.

[7] W. Geurts, F. Franssen, M. V. Swaaij, F. Catthoor, H. D. Man, and M. Moonen. Memory and data-path mapping for image and video applications. *application-driven architecture synthesis*, pages 143–166, 1993.

[8] E. D. Greef. *Storage Size Reductin for Multimedia Applications*. PhD thesis, ESAT/EE Dept., K.U.Leuven, 1998.

[9] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (Codes), pages 145–149, Seattle WA, USA, Mar. 1998.

[10] P. G. Kjeldsberg. *Storage requirement estimation and optimisation for data-intensive applications*. PhD thesis, Norwegian Univ. of Science and Technology, Trondheim, Norway, Mar. 2001. ISBN 82-7984-174-1.

[11] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas. Data dependency size estimation for use in memory optimization. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pages 908 –921, July 2003.

[12] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas. Detection of partially simultaneously alive signals in storage requirement estimation for data-intensive applications, 2003.

[13] A. W. Lim, S. W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. pages 103–112, 2001.

[14] N. Manjikian and T. S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, 1997.

[15] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Trans. on Computer-Aided Design*, 8(6):661–679, June 1989.

[16] P. Rydland, M. Palkovic, P. G. Kjeldsberg, E. Brockmeyer, and F. Catthoor. Inter in-place storage size requirment estimation. In *Norchip Conference*, pages –, Riga, LATVIA, Nov. 2003.

[17] Y. Song, R. Xu, C. Wang, and Z. Li. Data locality enhancement by memory reduction. In *International Conference on Supercomputing*, pages 50–64, Sorrento, Italy, 2001.

[18] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimensional incremental loop fusion for data locality. In Proc. IEEE Conf. on Application-Specific Systems, Architectures, and Processors, pages 14 –24, June 2003.

[19] Y. Zhao and S. Malik. Exact memory size estimation for array computation without loop unrolling. In Proc. 36th ACM/IEEE Design Automation Conf., pages 811–816, New Orleans LA, USA, June 1999.