Loop Fusion for Memory Space Optimization

Antoine Fraboulet Karen Kodary Anne Mignotte
Institut National des Sciences Appliquées de Lyon
20 avenue Albert Einstein
69621 Villeurbanne, France
£rstname.lastname@insa-lyon.fr

ABSTRACT

Portable or embedded systems as well as submicronic technologies have made the power consumption criterium crucial. Memory is known to be extremely power consuming. Moreover multimedia applications are memory intensive applications. Therefore, we propose new techniques to optimize a behavioral description of multimedia applications before the hardware/software partitioning (*Codesign*). These transformations are performed on "for" loops that constitute the main parts which handle the arrays of the multimedia code. This paper presents an optimal algorithm to reduce the use of temporary arrays by loop fusion. Although the algorithm is not polynomial, experiments have shown that it is very efficient.

1. INTRODUCTION

Code transformations for the design of an integrated system can be performed at several levels. For instance, Boolean functions minimization can be considered at the gate level. At the Register Transfer Level, transformations are performed on the control state charts by splitting or merging nodes. Moreover, at every stage of the design we can apply transformations used in software compilers [1, 2].

In this paper, we will focus on the design of data flow dominated embedded systems. These systems use signals and data stored in arrays such as images, video and sounds. This type of applications consumes memory for multidimensional data storage. More than a half of the surface of integrated systems of this kind is filled by memory. Memory is known to be power consuming. Therefore, this massive memory made power consumption criteria control compulsory. Power optimization by memory optimization can be done in several ways: the reduction of the size of the memory and improved data-movement strategies over the memory hierarchy.

Once the Hardware-Software partitioning is done, the memory has already been divided. It is therefore very important to make optimizations before this partitioning in order to deal with all the memory in homogeneous vision. Moreover, it allows to optimize both types of memories, the one that will be included in hardware and the one controlled by software.

Methods and tools defined in this paper are specific to Codesign [3] because in the case of software-only compilation the memory is already instantiated in hardware and cannot be tuned. Moreover, the memory is rarely shared in software; a memory cell is used for only one array cell (unless explicitly stated when the designer uses the same name for two non-overlapping objects).

On the opposite, during silicon compilation the physical memory is optimized all along the design chain. In-Place Mapping [4] can allow to share memory by overlapping arrays when possible, memory allocation can select memory modules upon several criteria (size, number of ports) and assignment computes hardware addresses for accessing arrays in memory [5, 6, 7]. The memory is instantiated on-demand and specific modules can be used or even built specifically. This will be the main assumption for memory optimization by code transformations.

All the optimizations undertaken during silicon compilation improve the design starting from a given description. However, a preprocessing source-to-source code transformation similar to the one used in software compilation can be applied on the design in order to improve the efficiency, or enable, further optimizations. The source-to-source transformations we propose are target independent code optimizations. We do not consider specific modeling of the target except that it has, at least, two levels of memory [8], including a cache memory. These transformations are good on general principles and are a *complement* to transformations that are designed for a specific target platform on which more precise information, such as cache line size or number of registers, can be used to drive further loop transformations [9, 10].

The handling of arrays is done mainly through "for" loops in multimedia applications. These loops form the critical part of the optimizations we want to apply at the Codesign stage. We propose to transform the algorithmic description of a design by exploiting and adapting techniques similar to the ones used in automatic parallelization [2] so as to reduce the size of needed memory and the associated power consumption by enabling powerful optimizations. Further optimizations have to be applied later in the design flow when more architectural parameters are set for the design in order to complete the optimization process. However,

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

these other steps and interactions between them are beyond the scope of this article.

In this paper, we will present a "for" loop transformation to optimize memory size. Loop fusion is a program transformation that collapses several loops into one. Memory minimization by loop fusion is obtained in our case by reducing the size of temporary arrays that are typically used to store intermediate results during multimedia processing.

An array written in a loop and read in another one must be stored in memory between these operations. In order to be removed from memory an array must be produced and used within the same loop nest and it must not be used elsewhere in the code. Once the statements that produce and consume values stored in an array have been merged within the same loop body several techniques can be used to reduce the size of this array or even to completely remove it.

For instance, *scalar replacement* [11] techniques can remove an entire array from the application's memory footprint as shown on Figure 1. *Scalar replacement* is a transformation that uses dependence information to find reuse of array values and expose it by replacing the references with scalar temporaries.

```
L1: for i=1 to n
   A[i] = ...
endfor
L2: for i=1 to n
   ... = f(A[i])
endfor
```

(a) source code

```
L12: for i=1 to n
   a = ...
   ... = f(a)
endfor
```

(b) after fusion

Figure 1: Scalar replacement after loop fusion

On Figure 1(a) the loop `L1` produces values stored in array `A` and the loop `L2` consumes these values. Merging these two loops (Figure 1(b)) allow to replace the array `A`, if not used elsewhere in the code, by a scalar `a` thus reducing the size of needed memory for the application.

Dependencies may not allow to remove completely an array by scalar replacement. In such cases we can apply intra-array storage order optimization [4]. This technique calculates an address reference window for a given storage order of a multi-dimensional array. Given the size of this window, we can "fold" array elements onto the same locations and hence increase the memory reuse. Intra array storage optimization is illustrated on Figure 2. The array `A` from source code on Figure 2(a) is replaced by a window of 2 values `a[0]` and `a[1]` on the fused code (Figure 2(b)).

```
L1: for i=1 to n
   A[i] = ...
endfor
L2: for i=1 to n
   ... = f(A[i-1],A[i])
endfor
```

(a) source code

```
L12: for i=1 to n
   a[i%2] = ...
   ... = f(a[i%2-1], a[i%2])
endfor
```

(b) after fusion

Figure 2: Intra-array storage optimization

Loop fusion increases the number of statements and accessed arrays within a loop nest. Sometimes loops access more data than can be handled by a cache. Further code transformations are needed to complete the optimization process once the *memory hierarchy parameters* such as the number of available registers or cache size and cache line size are known. In these cases, the iteration space of a loop can be blocked into sections whose reuse can be captured by the cache. Strip-mine-and-interchange [2] is a transformation that achieves this result. Its effect is to shorten the distance between the source and sink of a dependence so that it is more likely for data to reside in the cache when the reuse occurs. Extensive studies for cache optimizations can be found in [11, 9].

2. MEMORY MINIMIZATION BY LOOP FUSION

The algorithm we present in this section minimizes in an optimal way the size of the temporary arrays used in data dominated applications such as multimedia ones.

Previous algorithms for performing loop fusion have only considered the case when all loop headers are conformable [12, 13]. We consider a wider class of problems where the loops being considered for fusion need not have conformable headers. Loops with non conformable headers can be fused by using conditional statements to control the execution of operations within the loop nest. Both scalar replacement and intra array storage optimization techniques can handle conditional control flow.

The algorithm for maximal reuse by loop fusion proposed by McKinley and Kennedy in [12] has been proven NP-Hard. Our problem is different from *maximal reuse* because we do not constrain all edges to be maximaly fused but the complexity still remains exponential. We propose an algorithm that is efficient in practice and solves our problem with an optimal solution. A survey on loop fusion algorithms complexity can be found in [14].

2.1 Modeling the Problem

We approximate memory size requirement by the maximum size of time overlapping arrays. Memory used by non time overlapping arrays can be reused by inter storage optimization [4]. Our `Memory Cost` function is defined as following:

$$\text{MemoryCost} = \max_{\forall t} \left\{ \sum_{a_i \in \text{Live arrays}(t)} Size(a_i) \right\}$$

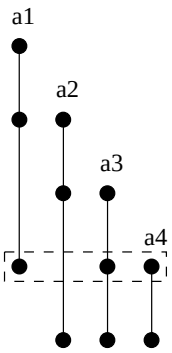
Where t is a time slot and $Size$ is the storage size of an array. Figure 3(b) represents the life time of arrays computed on Figure 3(a) source code. *MemoryCost* is maximized during loop `L4` (see figure 3(b)) and is equal to sum of the size of arrays `a1`, `a2`, `a3` and `a4`.

We use a *Data Flow Graph (DFG)* ($G = (V, E, A)$) representation for modeling the dependencies [2]. Graph nodes (V) represent the loop nests and edges (E) represent data dependences between these loops. A is the set of all arrays handled in the source code. Each array $a_i \in A$ has an associated weight $size(a_i)$.

A data dependence between two array references is represented by a hybrid distance/direction vector $\vec{\delta} = \{\delta_1 \dots \delta_n\}$ with the most precise information derivable. The vector component represents the data dependence corresponding left to right from the outermost loop to the innermost one enclosing the reference.

```
L1: for i=1 to n
    a1[i] = ...
endfor
L2: for i=1 to n
    a2[i] = f2(a1[i])
endfor
L3: for i=1 to n
    a3[i] = f3(a2[i+1])
endfor
L4: for i=1 to n
    a4[i] = f4(a1[i-1],a3[i])
endfor
L5: for i=1 to n
    ... = f5(a2[i],a3[i],a4[i])
endfor
```

(a) source code



(b) array life time

Figure 3: Array variables life time

Figure 4 represents the dependence graph computed from the source code on Figure 3(a). Loops `L2` and `L3` cannot be fused due to the dependence carried by `a2`. If the loops were merged the code would read `a2[i+1]` (from `L3`) before its computation (from `L2`) in the same iteration.

Two nodes can be merged if and only if none of the dependence are reversed in the fuse loop compared to the original code. An edge that carries a dependency which prevents the fusion of its source node and its destination node is called a *fusion preventing edge (FPE)* and is marked with a slash.

Each edge is labeled by the name of the array $a_i \in A$ that carries the dependence and is weighted by the size of this array. An edge is labeled by only one array and if there are multiple array dependencies between two loops then the graph becomes a multigraph with potentially multiple edges between two nodes.

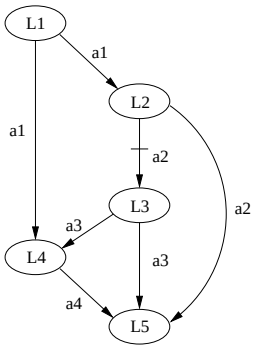


Figure 4: Modeling dependences

Isolated statements are also considered as nodes. Dependences between code statements and loops are preserved as we perform code reorganization during the transformation. An isolated statement will be represented in the graph as a regular node but all its incoming and outgoing edges will be marked as fusion-preventing ones.

An array can be removed from the memory, or at least minimized in size, if we can fuse all the loops that write into it with the loops that are reading its values. A removable array is marked with a star on the representations (see Figure 5).

2.2 Removable Arrays Detection

In order to remove an array from the program by fusion we need to fuse *all* the nodes connected by an edge which is labeled by this array in the DFG. An array cannot be removed by fusion there exists an edge $e = (u, v)$ labeled by this array and there exists a path from u to v in the DFG that contains a FPE.

In order to detect all the arrays that could be removed by loop fusion we perform a transitive closure on the DFG as can be seen on Figure 5.

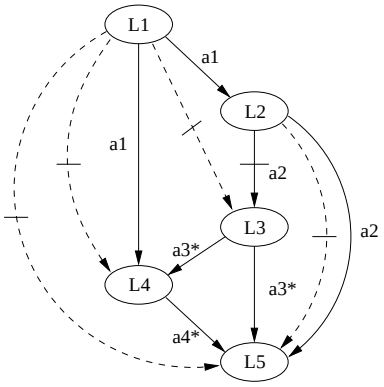


Figure 5: Transitive closure of the DFG for removable array detection

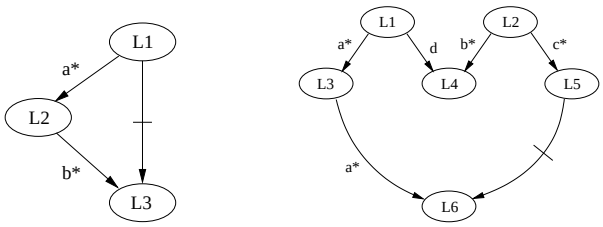
Array `a2` cannot be removed because there is a direct FPE between loops `L2` and `L3`. The same situation occurs for array `a1` where there is a path between `L1` and `L4` that goes through the FPE (`L2`,`L3`). Removing the array `a1` from the memory would require the fusion of loops `L1`, `L2` and `L4` and would create a cycle between the newly fused node `L124` and `L3` in the dependence graph. Such a cycle is not allowed in order to preserve the precedence constraints imposed by data dependencies.

Arrays `a3` and `a4` are marked with a star as they can be removed by merging loops `L3`, `L4` and `L5`. We call by extension *starred edges* an edge that carries a dependence on a removable array.

2.3 Con¤icts Detection and Resolution

The previous step can detect if an array can be removed from the memory by loop fusion but the detection is local and some problems can arise when we consider the removable arrays altogether. For instance, on Figure 6(a) arrays `a` and `b` can be removed if we consider them separately. Unfortunately, removing both at the same time is not feasible. The situation can be more complicated as can be seen on Figure 6(b) where the fusion cannot be performed without creating a dependence cycle between loops (`L1`,`L3`,`L6`) and loops (`L2`,`L4`,`L5`).

In order to complete the fusion process we need to solve all possible conflicts by reducing the set of starred arrays without compromising the global optimality. This resolution is done in two steps, the first one identifies all possible conflicts in the graph and the second one solves all these conflicts in a global optimal way.



(a) direct conflict (b) conflict path

Figure 6: Conflict between removable arrays

2.3.1 Con¤icts Detection

Problems arise when several potential removable arrays are located on a cycle of the graph while part of this cycle contains a FPE. Given μ an elementary cycle of the graph with a given direction over this cycle we denote μ^+ the set of edges in the cycle oriented toward the cycle cover direction, μ^- is the set of edges oriented the other way round (see [15] for more details). We can associate to μ a vector $\vec{\mu} = (\mu_1, \mu_2, \ldots \mu_{|E|})$ such as:

$$\mu_u = \begin{cases} +1 & \text{if } u \in \mu^+ \\ -1 & \text{if } u \in \mu^- \\ 0 & \text{if } u \notin \mu^+ \cup \mu^- \end{cases}$$

Note that $-\vec{\mu}$ is also a vector associated to the cycle μ with a different cover direction.

PROPOSITION 1. *In order to solve all possible problems that would create an illegal fusion we need to detect all undirected elementary cycle μ of the graph such as $\vec{\mu}$ is composed in the following way:*

- *all $+1$ (resp. -1) are starred edges*
- *at least one -1 (resp. $+1$) is a FPE*

PROOF. Assume that we have a fused graph G that is illegal. We want to prove that this original (unfused) graph was composed of, at least, one cycle such as described in Proposition 1. A fused graph G is illegal if it contains an oriented dependence cycle μ_d. This dependence cycle is composed of FPE or unstarred edges since all the starred edges have been fused. As there was no dependence cycle before the fusion (the original graph is a DAG) the last edge e_s that has been fused is starred and is oriented in a direction that prevents the original graph from being cyclic. Thus, the vector $\vec{\mu'}$ associated with the undirected cycle composed of $\mu \cup e_s$ corresponds to the definition given in proposition 1.

Now we want to prove that if there exists a non oriented cycle in the original graph that follows the proposition 1 the graph resulting from the fusion of all the starred edges will produce an illegal fused graph. Let us suppose that we have fused all but one starred edges from an undirected cycle μ to produce a graph G'. The vector $\vec{\mu'}$ in G' has only one $+1$ (resp. -1) that corresponds to a starred edge and at least one -1 (resp. $+1$) that corresponds to a FPE. If we fuse the last starred edge then we produce a directed cycle in the resulting graph composed of all the edges, of which at least one was an FPE, that were marked -1 (resp. $+1$) in $\vec{\mu'}$. □

Cycle detection algorithm. We perform cycle detection by exploring the graph for each edge $e = (u, v)$ that is fusion preventing by looking for all cycles from u to v that follows the proposition 1. This implies that we will look for all the cycles μ for which $\vec{\mu}$ contains only starred edges in μ^+.

In order to speed up the exploration we use the property that we can stop the exploration on a path μ as soon as we encounter an edge in μ^+ that is not starred.

```
exploration(G = (V, E),k,u,v)
begin
  mark[k] ← true
  for all t ∈ V do
  begin
    if t = v and t ≠ u then
      C ← C ∪ {v ∈ V|mark[v] = true}
    else if e = (k, t) ∈ E and e is starred
                and mark[t] = false
      exploration(G,t,u,v)
    else if e = (t, k) ∈ E and mark[t] = false
      exploration(G,t,u,v)
  end
  mark[k] ← false
end

begin
  C ← ∅
  for all e = (u, v) ∈ V
    exploration(C,u,u,v)
end
```

Figure 7: Graph exploration for conflict detection

We denote C the set of all cycles detected during the exploration of the graph. The next step will solve simultaneously all the dependency problems within these cycles.

2.3.2 Integer Linear Programming Con¤ict Resolution

In this section we present the ILP formulation [16] for solving dependency cycles conflicts detected in the previous step of the algorithm.

Once all the cycles have been detected we have to solve the global problem to decide which array will be taken out of the set of removable arrays.

We associate a binary variable x_{a_i} to each starred array a_i that could be removed but which has been included in a cycle during the previous step. If $x_{a_i} = 0$ then the array a_i will be considered for fusion otherwise ($x_{a_i} = 1$) the array will cease to be starred. For multi-graphs (for instance, k edges labeled $a_1, a_2, \ldots, a_k$) between two nodes u and v, a new variable x_{uv} is introduced to resume the arrays on this multi-edge. A variable x_{uv} will be set less or equal to each variable $x_{a_1} \ldots x_{a_k}$ associated with the arrays of the multi-edge. If x_{uv} is set to 1 (the multi-edge is removed from a path) then all associated variables will also be set to 1 and all arrays will be unstarred. Otherwise a variable x_{a_i} can be set to 1 without interfering with other arrays on the multi-edge (equation 3).

For each detected cycle $\mu \in C$ we need to decide which array carried by the edges in μ^+ will not be starred anymore. Thus the sum of all the variables x_{uv} in the μ^+ set of a cycle must be greater than or equal to 1 (equation 2).

The objective function of our ILP is thus to minimize the sum of the size of arrays that have to be removed from the set of all possible starred arrays detected in section 2.2 (equation 1).

$$\min \sum_{a_i \in A} size_{a_i} * x_{a_i} \quad (1)$$

$$\sum_{(u,v)\in\mu^+(c)} x_{uv} \geq 1, \qquad \forall \mu \in C \quad (2)$$

$$x_{uv} \leq x_{a_i}, \quad \forall a_i \in (u,v), \forall (u,v) \in E \quad (3)$$

$$x_{a_i} \in \{0,1\}, \qquad \forall a \in A \quad (4)$$

$$x_{uv} \in \{0,1\}, \qquad \forall (u,v) \in E \quad (5)$$

The ILP formulation given by (1), (2), (3), (4) and (5) minimizes the size that cannot be kept starred due to dependence constraints in a graph.

Values $x_{a_i} = 1, \forall i$ are always a feasible solution for the problem. Furthermore any feasible solution has a cost

$$\sum_{a_i \in A} size_{a_i} * x_{a_i} \geq 0$$

which ensures that an optimal solution always exists, because of this lower bound.

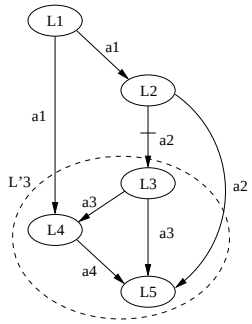
2.4 Graph Clustering and Fusion

All the edges that are still starred can now be fused. The only remaining step is to compute the clusters that will constitute the transformed dependence graph nodes. We have to ensure that if two nodes u and v will belong to the same cluster then all nodes that belong to a directed path from u to v will be also taken in the cluster. This step can be performed efficiently by computing a modified transitive closure in which if there is a path $u \xrightarrow{*} v$, a path $u \rightarrow w$ and a path $w \rightarrow v$ then u, v and w will be in the same cluster.

Code generation can be performed by writing the code for each loop following the numbering given by a simple order such as the height defined as follow:

$$h(x) = \begin{cases} 0 \text{ if } d^-(x) = 0 \\ \max\{h(y), y \rightarrow x \in E\} + 1 \text{ otherwise} \end{cases}$$

Figure 8 represents the modified dependence graph and the code corresponding to the loop fusion. Arrays `a3` and `a4` can be now replaced by scalars within the loop `L'3`.



(a) clustered dependence graph

```
L1: for i=1 to n
   a1[i]=...
endfor
L2: for i=1 to n
   a2[i]=f2(a1[i])
   endfor
L'3: for i=1 to n
   a3[i]=f3(a2[i+1])
   a4[i]=f4(a1[i-1],
            a3[i])
     ...=f5(a2[i],
            a3[i],a4[i])
   endfor
```

(b) modified code

Figure 8: Clustering and code output

3. EXPERIMENTATIONS

We have implemented this algorithm and tested it using randomly generated graphs. We have chosen to generate graphs that are a lot more complex than graphs that can be found in real applications. Generated graphs were ranging from 10 to 30 nodes for which the number of edges was twice the number of nodes. Each edge has the probability of 1/3 to be fusion preventing and all weights were randomly chosen between 1 and 100. Arrays were constrained to belong to a path (writing in an array cannot be done by two potentially parallel loops). The number of in-edges (edges pointing to a node) was limited to 10 for each node as well as the number of out-edges (edges going out of a node). This limit is very high compared to real application codes and plays a major role for the cycle enumeration problem.

Tests were done on 21,000 different graphs. We used the ILP solver `LP_SOLVE` freely available from its ftp site [17]. Although the cycle enumeration step is exponential in theory as well as the ILP resolution, we achieved an average calculation time of 0.050 seconds per graph (including cycle enumeration and ILP resolution) on a Pentium II running at 450 MHz. The average potential gain in size (arrays that were completely fused) was 60.7% and the average number of fused arrays was 61.8%.

4. FUTURE WORK AND CONCLUSION

We have presented in this paper an optimal algorithm to minimize the memory size needed by temporary arrays in an application. This algorithm has a theoretical exponential complexity due to its cycle enumeration step and its Integer Linear Program but is very efficient in practice. Furthermore, real life problem size are usually small as the number of nodes in our graph represents the number of loop nests in a program.

Our memory optimization cost function, unlike many others in the literature, does not rely on any memory hierarchy parameter and loop fusion for memory minimization can be applied as a general preprocessing step before any other optimization.

As it is a source to source transformation it can be easily integrated to an existing CAD tool-chain as a design preprocessor that can be run prior the traditional design flow. Thus, it will enable new optimizations obtained using existing memory and power management techniques in CAD tools. The power and memory interest of this automatic approach is on the one hand to reduce the design time by extracting optimizations for the description and on the other hand to improve the development quality by a tool which will propose interactive transformations that a designer could have missed.

We would like to combine the present work with our loop alignment techniques developed for memory accesses optimization [18]. Loop alignment can help to remove FPE from the original graph and can also reduce the dependence distance between statements. Loop fusion combined with loop alignment improves data locality within a loop nest. Once

a loop has been aligned, improved scalar replacement and intra array storage optimization results can be achieved over the data for both memory size and data locality.

5. REFERENCES

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman, *Compilers: Principes, Techniques, and Tools*, Addison-Wesley, 1986.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[3] Henry Chang, Larry Cooke, Merril Hunt, Grant Martin, Andrew McNelly, and Lee Todd, *Surviving the SOC Revolution: A Guide to Platform-Baased Design*, Kluwer Academic Publishers, 1999, ISBN 0–7923–8679–5.

[4] Eddy De Greef, *Storage Size Reduction for Multimedia Application*, Ph.D. thesis, Katholieke Universiteit Leuven – IMEC, Jan. 1998.

[5] M. Miranda, F. Catthoor, M. Janssen, and H.De Man, "Adopt: Efficient hardware address generation in distributed memory architectures," in *IEEE 9th International Symposium on System Synthesis, (ISSS'96)*, La Jolla, California, Nov. 1996, pp. 20–25.

[6] Preeti R. Panda, Nikil Dutt, and Alexandru Nicolau, "Architectural exploration and optimization of local memory in embedded systems," in *International Symposium on System Synthesis*, 1996.

[7] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau, *Memory Issues in Embedded Systems-On-Chip*, Kluwer Academic Publishers, 1999, ISBN 0–7923–8362–1.

[8] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, pp. 72–109, 1994.

[9] Chidamber Kulkarni, *Cache Optimization for Multimedia Applications*, Ph.D. thesis, Katholieke Universiteit Leuven – IMEC, Feb. 2001.

[10] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng, "Improving Data Locality with Loop Transformations," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 4, pp. 424–453, July 1996.

[11] Steve Carr, *Memory-Hierarchy Management*, Ph.D. thesis, Rice University, Feb. 1993.

[12] Kathryn S. McKinley and Ken Kennedy, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *The Sixth Annual Languages and Compiler for Parallelism Workshop*. 1993, pp. 301–320, Lecture Notes in Computer Science 768, Springer-Verlag.

[13] Guang R. Gao, Russ Olsen, Vivek Sarkar, and Radhika Thekkath, "Collective loop fusion for array contraction," in *1992 Workshop on Languages and Compilers for Parallel Computing*, New Haven, Conn., 1992, number 757, pp. 281–295, Berlin: Springer Verlag.

[14] Alain Darte, "On the complexity of loop fusion," *Parallel Computing*, vol. 26, no. 9, pp. 1175–1193, July 2000.

[15] M. Gondran and M. Minoux, *Graphs and Algorithms*, John Wiley, 1984.

[16] Alexander Schrijver, *Theory of Linear and Integer Programming*, John Wiley and Sons, New York, 1986.

[17] Michel Berkelaar, "Lp_solve Mixed Integer Linear Programming solver 3.2," Available at `ftp://ftp.es.ele.tue.nl/pub/lp_solve/`.

[18] Antoine Fraboulet, Guillaume Huard, and Anne Mignotte, "Loop Alignment for Memory Accesses Optimization," in *Twelfth International Symposium on System Synth esis Proceedings (ISSS'99)*. Nov. 1999, pp. 71–77, IEEE Computer Society Press.