Loop Alignment for Memory Accesses Optimization

Antoine Fraboulet
LIP, ÉNS Lyon
69364 Lyon Cedex 07
France
Antoine.Fraboulet@ens-lyon.fr

Guillaume Huard
LIP, ÉNS Lyon
69364 Lyon Cedex 07
France
Guillaume.Huard@ens-lyon.fr

Anne Mignotte
LIP, ÉNS Lyon
69364 Lyon Cedex 07
France
Anne.Mignotte@ens-lyon.fr

Abstract

*Portable or embedded systems allow more and more complex applications like multimedia today. These applications and submicronic technologies have made the power consumption criterium crucial. We propose new techniques thanks to which we can optimize the behavioral description of an integrated system before the hardware/software partitioning (*Codesign*). These transformations are performed on "for" loops that constitute the main parts of the multimedia code which handle the arrays. We present in this paper two new (polynomial) techniques for minimizing memory accesses in loop nests by data temporal locality optimization.*

1 Introduction

The design of embedded or integrated systems has become more and more complex, for instance with the appearance of multimedia and data dominated applications. This type of applications consumes a lot of memory for multidimensional data storage like images, sound or video. Thus more than a half of the surface of the integrated systems of this kind of application is filled by memory. This massive memory usage combined with submicronic technologies have made power consumption criteria control compulsory. Manual experimentations [2] have shown important consumption gains by code transformations on the algorithmic description of the design (MPEG4 experimentations have allowed a decrease of a factor 4 on average consumption and of a factor 10 on peak power). Experiments have also shown the relative cost of a memory operation compared to arithmetic computations (for example, a transfer from an external memory consumes 33 times more than a 16 bits addition).

Figure 1 shows where in the development flow global memory optimizations can be applied on a design. Once the Hardware/Software partitioning is done, the memory is already divided. It is therefore very important to make optimizations before this partitioning in order to deal with all the memory in homogeneous vision. We want here to optimize both types of memories, the one that will be included in hardware and the one controlled by software. The handling of data is done mainly through "for" loops in this kind of design. These loops form the critical part of the optimizations we want to apply at this stage. We thus propose to transform the algorithmic description of a design by using techniques similar to the ones used in automatic parallelisation [6, 7, 1] (see also [15] and [16]) so as to reduce the consumption in power and size due to memory.

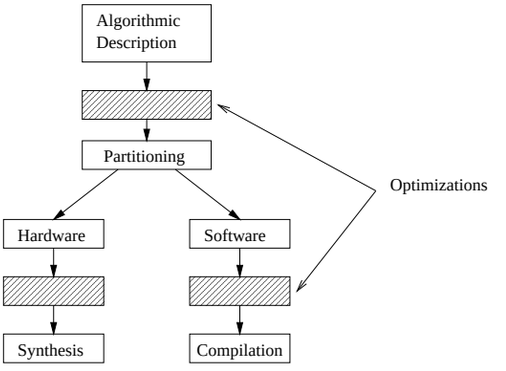


Figure 1. Loop transformations in the Codesign flow

2 Memory Optimization Criteria and Associated Techniques

Target architectures and applications impose a complex memory hierarchy: registers, hardware and/or software caches, on-chip or off-chip memory [3, 13]. A simplified view of the target architecture is shown on figure 2. The

power consumption of a memory access increases with the level from which the data has to be fetched. An access to an external memory consumes more power than an access to an on-chip memory. Memory hierarchies are well exploited if we can achieve a good data *temporal locality*. This locality represents the amount of time between two successive accesses to the same memory location (either write-read or read-read). At the level of abstraction at which we apply loop transformations, we can only represent this parameter in an abstract manner. We can see on the figure 3(a) a source

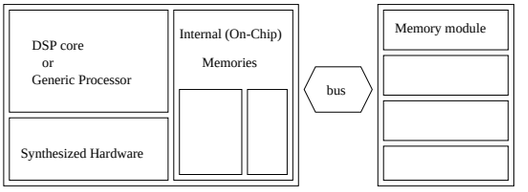


Figure 2. Target Architecture

```
for i=1,n
  b[i]=a[i]
for i=1,n
  c[i]=a[i+1]
for i=1,n
  d[i]=b[i-1]
```

(a) source code

```
for i=1,n
  b[i]=a[i]
  d[i]=b[i-1]
  c[i]=a[i+1]
end for
```

(c) loop merging

```
for i=1,n
  b[i]=a[i]
for i=1,n
  d[i]=b[i-1]
for i=1,n
  c[i]=a[i+1]
```

(b) moving code

```
d[1]=b[0]
for i=2,n
  b[i-1]=a[i-1]
  d[i]=b[i-1]
  c[i-1]=a[i-1]
end for
b[n]=a[n]
c[n]=a[n+1]
```

(d) loop alignment

Figure 3. Example of code transformations: moving, loop merging, loop alignment.

code composed with 3 different loops. The first loop computes the values stored in the array `b`, these values are then read in the third loop. We can have different approaches to measure temporal locality.

The first one would be to consider loops as atomic groups of instructions. As arrays are manipulated through loops, this implies that we do not consider locality between memory locations but we use a coarser grain represented by complete arrays. This level of granularity is used for *global* code transformations such as *code moving* or *loop merging*. The figure 3(b) shows the first transformation we can apply on the source 3(a) in order to improve temporal locality. The last loop has been shifted up in order to tighten the production and consumption of the array `b`. The next step is to merge the first two loops to have a common iteration space where the consumption of a value `b(i)` can be made nearer from its production. The figure 3(c) shows the three loops merged into one. The third loop has been also merged because it uses values from the array `a` which are also in use in the second loop. The second way to represent temporal locality is to look inside loops. This representation allows us to consider subsets of the arrays handled by the loop. This level of abstraction can be used to perform *local* loop transformations such as *interchange*, *skewing*, *folding* or *alignment*. On the example 3(c) for each iteration, the loop produces the value `b[i]`, `c[i]`, `d[i]` and uses the values `b[i-1]`, `a[i]` and `a[i+1]`. The next transformation step will take into account values that are produced and consumed in different iterations of the loop. We call this gap of iterations a *distance*. A new value is produced at each iteration and must be kept into a separate foreground (on-chip) memory buffer until its last use by another statement of the loop. The number of "memories" needed to store a value computed and used in different iterations is given by the amount of iterations the value has to cross.

Figure 3(d) shows the loop once aligned to optimize the use of the arrays `b` and `a`. We can see that values of the array `b` are consumed as soon as they are produced. This optimization increases the probability we have to find the value `b[i-1]` in a very high level of the memory hierarchy. Optimization has also been performed in the use of the array `a`: the value `a[i-1]` has to be fetched from distant memory only once per loop iteration. We will use and develop this measure of temporal locality for loop alignment in the next section.

Memory is by itself a source of power consumption. It is also important to reduce the size of the memory needed by an application. A reduction of the amount of needed memory can decrease the number of levels in the memory hierarchy. A significant reduction would ideally allow to store everything in the on-chip memory, thus enabling the removal of the off-chip memory. This optimization can be done only if the consumption of a value appears right after its production. The array `b` on figure 3(d) can be completely removed from memory if the array is not used elsewhere in the code. This optimization of memory size is also associated with loop alignment.

Loop transformations at this stage of the codesign flow can do a lot by themselves. But they cannot perform all the needed transformations. More powerful optimizations—in terms of power and memory size gain—can be achieved in

later steps of the compiling flow, once the design has been partitioned. Optimizations like *in-place mapping* [4], *memory distribution* across modules [13], *cache level* optimizations [11] and many others [3, 13] have to be done afterwards. These optimizations are *enabled* by high level transformations done *before* the hardware-software partitioning. Optimization criteria developed in the next section have been defined considering that these transformations are performed afterwards.

We present in the next two sections 3 and 4 two different algorithms for memory accesses optimizations. The first one minimizes the number of buffers needed between iterations of a loop by loop alignment. The second one finds a minimal bound for all dependencies of a loop.

3 Buffers Minimization in Loop Nests by Loop Alignment

The algorithm we present in this section minimizes the size of the foreground memory needed to store values that are computed and used in the same loop. This minimization can also be seen as optimizing the average distance in terms of temporal locality between read and write accesses to the same variable in different loop iterations. This technique is not only useful to keep values in a memory near the top of the hierarchy (where memories are smaller and less power consuming) but it can also decrease the memory size needed by the application (a dimension of an array can be reduced to a scalar value for example).

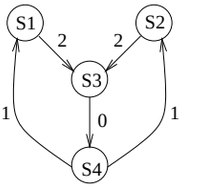
3.1 Modeling the Problem for a Single Loop (Monodimensional Case):

We use a Reduced Dependence Graph ($G = (V, E, w)$) representation for modeling the problem. Graph nodes (V) represent the statements of the loop, edges (E) represent data dependences between these statements. Each dependence edge e is weighted by a distance w_e which corresponds to the number of iterations between the two accesses. These distances are positive as a program cannot use a value before its computation. We restrict ourselves to the case of uniform (constant) forward (write-read) dependences over the loop to be able to use *retiming* [12] techniques.

Example: we can see on figure 4 that there are two dependences of distance 2 in the first loop to statement `S3` from statement `S1` and `S2`. There is also a dependence of distance 1 in the inner loop from statement `S4` to statements `S1` and `S2`. The value produced by the statement `S3` (`c[i]`) is consumed in the same iteration by the statement `S4`. There is a dependence distance of 0 between these last two statements.

```
  for i=1,n
S1: a[i]=2*d[i-1]
S2: b[i]=3*d[i-1]
S3: c[i]=a[i-2]+
         b[i-2]
S4: d[i]=2*c[i]
  end for;
```

(a) source code



(b) dependence graph

Figure 4. Modeling dependences in a loop

The number of buffers needed for a statement represented by a node u depends on the dependence length w_e of all its out-edges e. This amount is given by the following relation.

Cost per node:

$$c_u = \max_{e=(u,v)\in E} w_e \qquad (u \xrightarrow{e} v)$$

The total number of buffers across iterations in the graph is thus:

$$Cost(G) = \sum_{u\in V} c_u.$$

Minimizing $Cost(G)$ can be solved in polynomial time, as we will see in the section 3.3, by using *retiming* techniques. A retiming value r_u (integer) is associated with each node u. This weight represents a shift (or a delay) in a number of iterations for the associated statement. Therefore applying a retiming on a graph modifies dependence distances. The graph after retiming can be rewritten into a code, functionally equivalent, but with new dependence distances $w_{r,e}$ given by the following relation:

$$w_{r,e} = w_e + r_v - r_u, \qquad (u \xrightarrow{e} v)$$

We must define a constraint in order to obtain a *legal* retiming on the graph, dependence distances after retiming must be positive (we cannot use a value before it is computed)

$$w_{r,e} \geq 0, \ \forall e \in E$$

3.2 Integer Linear Program Formulation:

In this section we present the ILP formulation for the problem of minimizing $Cost(G)$.

$$\min \sum_{u\in V} c_u \tag{1}$$

$$w_e + r_v - r_u \geq 0, \quad \forall e = (u, v) \in E \tag{2}$$

$$c_u \geq w_e + r_v - r_u, \quad \forall e = (u, v) \in E \tag{3}$$

The objective function of our ILP formulation is given by the relation (1). The constraints (2) ensure that we have a legal retiming. The cost of a node after retiming is given by the inequation (3). As we cannot use a $\max$ function in the constraint—the problem would not be linear—we must define the cost of a node u to be greater or equal to the cost of each out-edge. Minimizing (1) ensures that the maximal value is reached by c_u, giving the expected cost for each node.

The ILP formulation given by (1), (2) and (3) minimizes the number of buffers needed across iterations of the loop.

Values $r = 0$ and $c_u = \max_{e=(u,v)\in E} w_e$ are always a feasible solution for the problem. Furthermore any feasible solution has a cost $\sum_{u\in V} c_u \geq 0$ which ensures that an optimal solution always exists, because of this lower bound.

Size of the formulation:

- variables (r_u and c_u): $2|V|$;
- constraints: $2|E|$.

3.3 A Polynomial Algorithm

We denote $C = \{c_u\}$ the row vector of node costs, $R = \{r_u\}$ the row vector of node retiming values and $W = \{w_e\}$ the row vector of edge weights. The matrix representation of the previous ILP formulation is given by the relation (4).

$$\min \left\{ \begin{pmatrix} R & C \end{pmatrix} . \begin{pmatrix} 0 & 1 \end{pmatrix} \middle| \begin{pmatrix} R & C \end{pmatrix} \begin{pmatrix} -A & A \\ 0 & A^+ \end{pmatrix} \geq \begin{pmatrix} -W & W \end{pmatrix} \right\}, \tag{4}$$

where the matrix A is the nodes-edges incidence matrix (each column has one and only one +1 and -1, see [10]) of the reduced dependence graph $G = (V, E, w)$ and the matrix A^+ is defined as follow:

$$\begin{cases} a^+_{i,j} = 1 \text{ if } a_{i,j} = 1 \\ a^+_{i,j} = 0 \text{ otherwise} \end{cases} \tag{5}$$

where the $a_{i,j}$ are the elements of A (the matrix A^+ has the same dimensions as A but we keep only its positive values). This corresponds to the fact that we define the cost c_u only for out-edges of a node and not for in-edges.

The matrix $M = \begin{pmatrix} -A & A \\ 0 & A^+ \end{pmatrix}$ can be transformed into the matrix $M' = \begin{pmatrix} -A & A^- \\ 0 & A^+ \end{pmatrix}$ by a unimodular transformation (by subtracting the last $|V|$ rows from the first $|V|$). The matrix M' is a nodes-edges incidence matrix and is totally unimodular [10] (every square regular submatrix of M' has a determinant of -1, 0 or +1). As we have transformed M to M' by a unimodular operation the matrix M is also totally unimodular and we can conclude that the ILP formulation (4) admits an integral optimal solution in the rationals and that it can be solved by a polynomial algorithm [14, 5].

In practice, the size of the ILP is likely to be small (proportional to the number of statements in a loop that have array accesses). Although this problem can be solved very efficiently by any ILP solver we will show how to use the dual form of the problem (4) to reduce it to a *minimal cost flow* problem [14].

Interpretation of the Dual Problem: the dual form of the problem (4) is given by the problem (6) [14, 5]. X and Y are row vectors of size $|E|$ representing the new set of unknowns x_e and y_e.

$$\max \left\{ \begin{pmatrix} -W & W \end{pmatrix} \begin{pmatrix} X & Y \end{pmatrix} \middle| M \begin{pmatrix} X & Y \end{pmatrix}^t = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} X & Y \end{pmatrix} \geq 0 \right\} \tag{6}$$

We first change the problem to have a minimization problem instead of a maximization one. The transformed problem is given in equation (7).

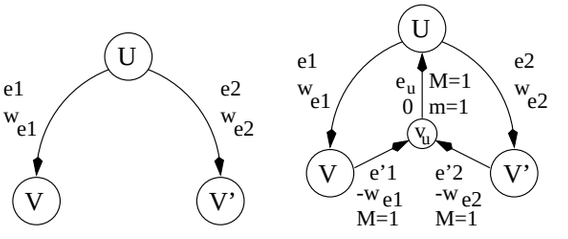
$$\min \left\{ \begin{pmatrix} W & -W \end{pmatrix} \begin{pmatrix} X & Y \end{pmatrix} \middle| M \begin{pmatrix} X & Y \end{pmatrix}^t = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} X & Y \end{pmatrix} \geq 0 \right\} \tag{7}$$

The cost function on the unknown variables x_e and y_e to minimize is the scalar product $W.(X - Y)$. This minimization is controlled by two sets of constraints which are given in equations (8) and (9).

$$A.(X - Y)^t = 0, \ \begin{pmatrix} X & Y \end{pmatrix} \geq 0 \tag{8}$$

$$A^+.Y^t = 1, \ Y \geq 0 \tag{9}$$

The first set of constraints (8) imposes that $(X - Y)$ be a flow over the graph [10]. The other set of constraints given by the equation (9) means that the Y part of the flow on a node must be directed through one and only one of the out-edges of this node. These constraints can be taken into account by constructing a new graph $G'(V', E', w')$ from $G(V, E, w)$ in the following way: edges of the orig-



(a) original graph (b) modified graph

Figure 5. Graph transformations for flow algorithm resolution

inal graph G are kept in the transformed graph G' with their respective weight. We then introduce *virtual nodes* $\{v_u\}$. For each edge $e = (u, v)$, we build an edge $e' = (v, v_u)$. The edge e' is weighted by $-w_e$ and has a maximal flow capacity M_e set to 1. Another edge is added from the virtual node v_u to the node u. This edge has both minimal m and maximal M flow capacity set to 1 and is null weighted.

An example of transformation for a node with two out-edges is given on figure 5.

Let f be a flow of G', we define for each edge e a couple (x_e, y_e) in the following way:

$$x_e = f(e) \tag{10}$$

$$y_e = f(e') \tag{11}$$

Proposition 1 *There is a bijection between the flows of G' and the feasible solutions of the dual problem. Furthermore minimal cost flows of G' correspond to optimal solutions for the dual problem.*

Proof: A complete proof is available in [9]

Computing a Minimum cost flow f on G': we use a standard algorithm for computing minimum cost flows with both capacity and lower bounds (see [5]). To start the algorithm we trivially construct an admissible flow for G' satisfying capacities on edges (v_u, u) by choosing for each vertex u an arbitrary outgoing edge (u, v) and putting some flow through $\{(u, v), (v, v_u), (v_u, u)\}$. The minimal cost flow algorithm introduces a graph $R^*(f)$ built from the flow f that will be used in the next paragraph.

Solution of the primal problem from the dual one: once we have found an optimal solution for the flow problem we have to compute the corresponding retiming for the primal problem.

We construct the retiming in the following way: we consider the optimal flow with its associated graph $R^*(f)$. We add a source S with a null weighted edge to all nodes of $R^*(f)$ and we compute the shortest path π_u from S to each node $u' \in V'$ by a Bellman-Ford algorithm [10].

We chose for each node u

$$r_u = -\pi_u$$

as a retiming value and the cost

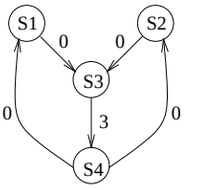
$$c_u = \max_{e=(u,v)\in E} w_e + r_v - r_u.$$

Proposition 2 *The proposed solution is feasible and is optimal for the primal problem.*

Proof: A complete proof is available in [9]

Complexity: The complexity of the algorithm is $O(|V||E|.(\sum_{e\in E} w(e)))$

The example we gave on figure 4, which needs 5 "buffers" is optimized on figure 6 with a cost of only 3 memories.



transformed code

prologue

```
  for i=2,n-2
S4: d[i-1]=2*c[i-1]
S1: a[i]=2*d[i-1]
S2: b[i]=3*d[i-1]
S3: c[i+2]=a[i]+b[i]
  end for
```

epilogue

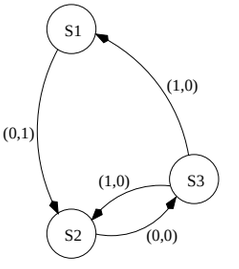
Figure 6. Example of the figure 4 after iteration buffers minimization

3.4 Extending the Problem to the Multidimensional Case

In this section we are extending the problem to deal with loop nests. In the multidimensional case, dependences are handled by integer vectors. A component w_i of a dependence vector w corresponds to the distance carried by the i^{th} loop in the nest, starting from the outer loop. A loop nest composed of n loops will thus carry dependence vectors with at most n components (it can be less than n if the nest is not perfectly nested). Figure 7 shows the graph representation for the multidimensional case.

```
for i = 1,n
  for j = 1,m
S1: a[i,j]=c[i-1,j]
S2: b[i,j]=a[i-1,j]+
           c[i-1,j]
S3: c[i,j]=b[i,j]
  end for
end for
```

(a) source code



(b) dependence graph

Figure 7. Modeling dependences in a loop nest

In this case, to be correct, the dependences have to be lexico-positive. We denote by $\geq_{lex}$ the lexicographic order.

The equation (2) taken from the monodimensional case now becomes:

$$w_e + r_v - r_u \geq_{lex} 0,\ \forall e = (u, v) \in E \qquad (12)$$

These constraints are not linear and they cannot be linearized, to our knowledge, without losing total unimodularity on the matrix.

We propose here an efficient heuristic solution for the multidimensional problem by reducing it to the monodimensional one. This reduction is done by applying the monodimensional algorithm several times on the loop nest. Dependences handled by external loops are the more expensive ones as they imply manipulations of complete subarrays and also imply longer life time for the buffers we want to minimize, so we will consider them first. Incremental optimizations, like the one we propose, also provide the opportunity to stop optimizing memory accesses and size given a tradeoff in order to switch to another optimization problem. For example, memory minimization is often dual with maximizing parallelism and finding an absolute optimal on memory may lead to very poor parallelism detection in the next step of the compilation.

Heuristic for the Multidimensional Case: the heuristic we propose for memory accesses in loop nests consist in transforming the nest loop by loop starting from the outermost loop to the innermost one. At each step, we dispose of a graph $G = (V, E, w)$ of dimension n, and we apply our algorithm in the first dimension to find a retiming r that optimizes it. Then we define $\tilde{G}_r = (V, E', w')$ from G_r as follow:

$$E' = \{e \in E \,|w_e \leq_{lex} (0, +\infty, \ldots, +\infty)\}.$$

and w' is defined from w by restricting it to its $n-1$ last components. Finally we proceed to the next step with $\tilde{G}_r$.

Proposition 3 *This heuristic produces correct code after retiming.*

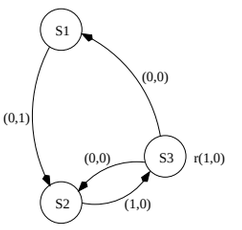
Proof: A complete proof is available in [9]

Note: if the optimization is not done down to the innermost loop, a final retiming is needed on last dimensions in order to ensure correction (see proof in [9] for more details).

Figure 8 shows the optimized code for example on figure 7 where the first loop has been aligned.

4 Bounding the maximal distance

Minimizing buffers between loop iterations as we have seen in the previous section can increase the dependence distance for some variables while decreasing for others.



transformed code

prologue

```
for i = 2,n-1
  for j = 1,m
S3: c[i-1,j]=b[i-1,j]
S1: a[i,j]=c[i-1,j]
S2: b[i,j]=a[i-1,j]+
          c[i-1,j]
  end for
end for
```

epilogue

Figure 8. Example of the figure 7 after iteration buffers minimization

However it might be more suitable to make dependences more regular to fit better particular hardware design constraints (such as known number and size of cache lines). The polynomial algorithm we propose here can modify the dependence distances of a program by retiming [12] in order to find a minimal distance bound.

Given the reduced dependence graph $G = (V, E, w)$ of a loop nest as described in section 3.1 we can bound the maximal dependence distance k of the graph.

Problem: Let S be a set of $|V|$ inequalities of the form

$$w_e + r_v - r_u \quad \leq \quad k,\ \forall e = (u, v) \in E \qquad (13)$$

on the unknown retiming values $r_u, u \in V$. These inequalities represent the dependence distances once the retiming is applied to the graph where each distance must be lower or equal to a fixed value k. The problem is to determine feasible values for the r_u or determine that the system is inconsistent.

Let $a_e = k - w_e$ for all e in E, the system (13) is transformed into the following system:

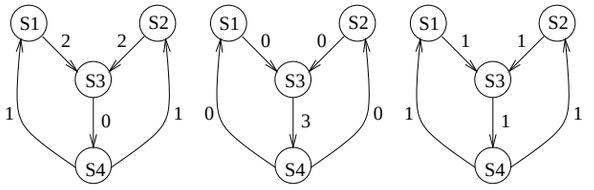
$$r_v - r_u \quad \leq \quad a_e,\ \forall e = (u, v) \in E. \qquad (14)$$

Such system in which each constraint has the form of inequality (14) arises in the shortest path problem (see [10]) that has been extensively studied and can be solved—or determined inconsistent—in $O(|V||E|)$ time by a Bellman-Ford algorithm [10].

The minimal solution for our problem is obtained with a logarithmic binary search for the minimal k in $[0, \max_{e \in E}(w_e)]$ Complexity: each Bellman-Ford verification can be computed in $O(|V||E|)$, this verification is used $O(\ln(\max_{e \in E}(w_e))$ times during the binary search. The complexity of the problem is bounded by $O(|V||E| \ln(\max_{e \in E}(w_e))$.

Starting from the graph 9(a), minimizing the maximal dependence distance of the graph produces the graph 9(c)

for which the needed foreground memories are equal to 4. A better solution, as regards memory size, would have been achieved by the solution 9(b) for which only 3 foreground memories are needed. However the solution 9(c) is more regular and may be best suited for specific architectures.



(a) source

(b) minimizing iteration buffers (section 3)

(c) minimizing maximal distance

Figure 9. Minimizing the maximal distance

5 Future Work and Conclusion

We have presented in this paper a polynomial algorithm for memory accesses optimization by loop alignment (folding) in the monodimensional case and a heuristic based on this algorithm for the multidimensional case. A second polynomial algorithm was presented to minimize the maximal dependence distance of a loop nest.

These algorithms will be extended to deal with conditional execution ("if") in order to be able to model real applications. Therefore we want to use the *Program Dependence Graph* defined in [8] by Ferrante et al. to take into account both data flow and control flow dependencies for source to source loop transformations.

We have also presented the approach we have taken for automatic loop transformations on data dominated applications in multimedia applications. The 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 proposing interactive transformations that a designer could have missed.

We want to go further in the development of new global loop transformation techniques (loop merging, code moving, ...) as well as local transformations (loop interchange, loop alignment, skewing). These techniques will be integrated in our transformation engine LOOPING [17]. The LOOPING project we have started is a transformation engine prototype for source to source transformations of data dominated applications in portable or embedded systems geared toward memory and power consumption. This engine has to be both automatic and interactive because there are many tradeoffs that only the designer of a system can control at this level of transformations.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[2] E. Brockmeyer. Low power data transfer and storage exploration for mpeg-4 on multi-media processors. Master's thesis, IMEC, Apr. 1998.

[3] F. Catthoor. Power-efficient data storage and transfer methodologies: current solutions and remaining problems. In *CS annual rush on VLSI*, Orlando, Apr. 1998.

[4] E. De Greef. *Storage Size Reduction for Multimedia Application*. Phd thesis, IMEC, Jan. 1998.

[5] D. de Werra. *Eléments de programmation linéaire avec applications aux graphes*. Presses polytechniques romandes, 1 edition, 1990. ISBN 2-88074-176-9.

[6] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part I, One Dimensional Time. *Int. J. of Parallel Programming*, 21(5), Oct. 1992.

[7] P. Feautrier. Fine-grain scheduling under resource constraints. In *7th Workshop on Language and Compiler for Parallel Computers*, Cornell University, Aug. 1994. to appear in LNCS.

[8] J. Ferrante, K. J. Otteinstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] A. Fraboulet, G. Huard, and A. Mignotte. Loop Alignment for Memory Accesses Optimization. Research Report 1999–26, École Normale Supérieure de Lyon, Apr. 1999. available at ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1999/RR1999-26.ps.Z.

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

[11] C. Kulkarni, F. Catthoor, and H. D. Man. Cache optimization for multimedia compilation on embedded processors for low power. In *Proc. Intnl. Parallel Proc. Symp.(IPPS)*, pages 292–297, Orlando FA, Apr. 1998.

[12] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. In *Algorithmica*, volume 6, pages 5–35. Springer-Verlag, 1991.

[13] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, 1999. ISBN 0–7923–8362–1.

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

[15] PRiSM SCPDP Team. Systematic construction of parallel and distributed programs. World Wide Web document, http://www.prism.uvsq.fr/english/parallel/paf/autom_us.html.

[16] Stanford Compiler Group. Suif compiler system. World Wide Web document, http://suif.stanford.edu/suif/suif.html.

[17] LOOPING Project. http://www.ens-lyon.fr/∼afraboul/looping/.