Source Code Loop Transformations for Memory Hierarchy Optimizations

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

Abstract

Portable or embedded systems allow complex applications like multimedia today. These memory intensive applications and submicronic technologies have made the power consumption criterion crucial. We propose new source to source transformations thanks to which we can optimize the behavior of these applications by reducing the amount of needed physical memory and hence the associated power consumption. 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 new techniques for minimizing memory size by loop fusion and loop alignment. These techniques do not depend on any architectural consideration or parameter as we do not use cache size or cache line size to drive the transformations. Further optimizations that will consider these parameters to improve data-movement strategies over the memory hierarchy can be applied later to complete and refine the optimization framework.

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. More than 50% of the surface of integrated systems of this kind of application is filled by memory. This massive memory usage combined with submicronic technologies have made power consumption criterion control compulsory. Manual experimentations [2] have shown important memory size and power 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). Thus it has become very important to minimize redundant data transfers over the memory hierarchy as well as the application memory footprint.

Once the Hardware/Software partitioning within the design 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. The main drawback is that we do not have any information about the memory hierarchy structure and physical specifications that will be available on the final implementation. The only assumption here is that the system will include a memory hierarchy with at least two levels of memory called background (main) and foreground (caches) memories. We want here to optimize both types of memories either if they are controlled by hardware or controlled by software.

The handling of data is done mainly through "for" loops in multimedia designs. 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 parallelization [1] so as to reduce the memory size and memory accesses power consumption [14, 2, 10, 5]. Techniques we propose do not depend on any architectural consideration or parameter except that there exists a memory hierarchy. They are well suited for embedded or portable systems design space exploration prior to the Codesign step [15, 4].

The next section will present the optimization criteria we use for the transformations. Sections 3 and 4 will present two different optimizations, namely loop fusion for memory minimization and loop alignment for buffer minimization.

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 memories [15]. The power consumption of a memory access increases with the level from which the data has to be fetched. As 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). Temporal locality is improved as this amount of time gets shorter. At the level of abstraction at which we apply loop transformations, we can only represent this parameter in an abstract manner. We can have different approaches to measure temporal locality.

2.1. Coarse Grain Locality Optimization

The first approach considers loops as atomic groups of instructions. As arrays are manipulated through loops, this implies that we do not consider locality between exact 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 fusion* [1]. For instance, the last loop on figure 1(a) has been shifted up on figure 1(b) by code moving in order to tighten the production and consumption of the array `b`. Loop fusion on figure 1(c) gives a common iteration space where the consumption of a value `b[i]` can be made nearer from its production. We will use and develop the coarse grain temporal locality for *memory minimization by loop fusion* in section 3.

2.2. Fine Grain Locality Optimization

The second approach 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* [1]. On the example on figure 1(c) the loop produces, for each iteration, a value

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

(b) moving code

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

(c) loop fusion

```
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 1. Example of code transformations: moving, loop merging, loop alignment.

`b[i]`, `c[i]`, `d[i]` and uses values `b[i-1]`, `a[i]` and `a[i+1]`. A new value `b[i]` 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. This iteration gap is called a *distance*.

Figure 1(d) shows the loop once aligned to optimize the use of the arrays `a` and `b`. 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 the fine grain temporal locality for *buffer minimization by loop alignment* in section 4.

2.3. Optimization Flow and Related Works

Memory accesses are by themselves a huge source of power consumption. It is also important to reduce the size of the memory needed by an application. A significant reduction of the amount of needed memory can decrease the number of levels in the memory hierarchy and it could ideally allow to store everything within 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. For instance, the array `b` on figure 1(d) that is produced and used within the same loop can be completely removed from memory and replaced by a scalar if it is not used elsewhere in the code [3, 6].

Several efforts have been aimed at using loop transformations and scheduling techniques to improve locality by loop transformations. McKinley and Kennedy in [13] studied maximum reuse by loop fusion and proved the problem to be NP-Hard. Despite our problem is different from maximum reuse, as we are interested in fusing dependencies that have a same label and not constrain all dependencies to be maximally fused, the complexity of our loop fusion problem still remains exponential. Other global loop transformation techniques for memory size optimizations [12, 3] are driven by memory hierarchy parameters such as cache size, cache line size or the number of available registers.

Once the partitioning between hardware and software has been done many optimizations can be applied on a design. A recent study on cache level optimization techniques can be found in [10] and data layout organization is studied in [6]. These optimizations are *enabled* by high level transformations done *before* the hardware-software partitioning. Optimizations developed in the two next sections have been defined considering that these transformations are performed afterwards.

3. Loop Fusion for Memory Minimization

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. The size of the memory is approximated by the maximum size of time overlapping arrays. This is based on the assumption that the memory will be shared afterward by further optimization techniques.

We use a data flow graph (DFG) ($G = (V, E, A)$) representation for modeling the dependencies [1]. 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 $size(a_i)$. Our `Memory Cost` function is defined as following:

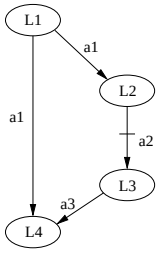
$$\text{MemoryCost} = \max_{\forall t} \left\{ \sum_{a_i \in \text{Live arrays}(t)} size(a_i) \right\}$$

Where t is a execution time slot and $size$ is the storage size of an array.

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 multiple edges between two nodes. Figure 2(b) represents the dependence graph

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

(a) source code



(b) dependence graph

Figure 2. Modeling dependences

computed from the source code on Figure 2(a). Loops `L2` and `L3` cannot be fused due to the dependence carried by `a2`. If these loops were merged the code would read `a2[i+1]` (from `L3`) before its computation (from `L2`) in the same iteration. 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.

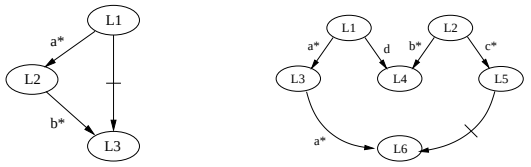
3.1. Removable Arrays Detection

In order to remove an array from the program by merging the loops we need to merge all the nodes connected by an edge which is labeled by this array in the DFG. Merging is not needed for an edge $e = (u, v)$ labeled by an array `a` if there exist a path from u to v that contains a FPE. Detection of all the arrays that could be removed by loop fusion is performed with a *transitive closure on the DFG* ($O(|V|^3)$). For instance, array `a1` cannot be removed because 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 `L124` and `L3` in the dependence graph. Such a cycle is not allowed in order to preserve the precedence constraints imposed by data dependencies. Removable arrays are marked with a star as they can be removed by merging loops. We call by extension *starred edges* an edge that carries a dependence on a removable array.

3.2. 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 removable arrays altogether. For instance, on Figure 3(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 3(b) where the fusion cannot be performed without creating a dependence cycle between loops (`L1`,`L3`,`L6`) and loops (`L2`,`L4`,`L5`).



(a) direct conflict (b) conflict path

Figure 3. Con¤ict Between Potentially Removable Arrays

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.

Conflicts 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 [9] for more details).

We can associate to μ a vector $\vec{\mu} = (\mu_1, \mu_2, \dots \mu_{|E|})$ 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 and cut 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*

A complete proof is available in [7]. Cycle which follows the proposition 1 are detected by exploring the graph for each edge $e = (u, v)$ that is fusion preventing. This implies that we will look for all the cycles μ for which $\vec{\mu}$ contains only starred edges in μ^+.

Once the set C of all the cycles have been detected we have to decide which starred array will not be considered for fusion. This step is done through an *Integer Linear Program (ILP)* [16].

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 removal by fusion otherwise ($x_{a_i} = 1$) the array will cease to be starred. For multi-graphs (for instance, k edges $a_1, a_2, \dots, a_k$) between two nodes u and v, a new variable x_{uv} is introduced to resume the arrays on this multi-egde. A variable x_{uv} will be set less or equal to each variable $x_{a_1} \dots x_{a_k}$ associated with 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 arrays will be unstarred. Otherwise a variable x_{a_i} can be set to 1 without interfering with other arrays on the multi-edge. For each detected cycle $\mu \in C$ we need to decide which array carried by edges in μ^+ will not be starred anymore. Thus the sum of all the variables x_{uv} on a cycle path must be greater than or equal to 1.

The objective function of our ILP is thus to minimize the sum of the size of starred arrays that we have to remove from the set of all possible starred arrays we found in section 3.1 (equation 1).

$$\min \sum_{a_i \in A} size_{a_i} * x_{a_i} \tag{1}$$

$$x_{a_i} \in \{0,1\}, \qquad \forall a \in A \tag{2}$$

$$\sum_{(u,v) \in \mu^+(c)} x_{uv} \geq 1, \qquad \forall \mu \in C \tag{3}$$

$$x_{uv} \in \{0,1\}, \qquad \forall (u,v) \in E \tag{4}$$

$$x_{uv} \leq x_{a_i}, \quad \forall a_i \in (u,v), \forall (u,v) \in E \tag{5}$$

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.

3.3. Graph Clustering and Fusion

All edges that are still starred can now be fused. The only remaining step is to compute the clusters that will make 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* ($O(|V|^3)$) 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 (see [7] for details). Code generation can be performed by writing the code for each loop following the numbering given by a simple order such as the node height.

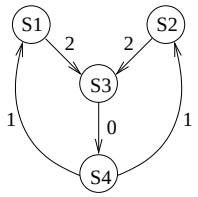
4. Buffers Minimization by Loop Alignment

Once the coarse grain temporal locality has been considered as in the previous section, the fine grain locality can be optimized using more precise dependency information. The algorithm we present in this section minimizes, in polynomial time, the size of the foreground scalar memories needed to store values that are computed and used within 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), it reduces the register pressure and can also decrease the memory size needed by the application (a dimension of an array can be reduced to a scalar value for instance [3, 6]). We present here only the mono-dimensional case (i.e. single loop) due to the lack of space. The multi-dimensional case is not polynomial anymore and a heuristic is proposed in [8].

We use a Reduced Dependence Graph ($G = (V, E, w)$) representation for modeling the problem. Graph nodes (V) represent the loop statements, 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* techniques [11].

```
  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

We can see on figure 4 that there are two dependences of distance 2 in the first loop to statement `S3` from statements `S1` (`a[i-2]`) and `S2` (`b[-i2]`). There is also a dependence of distance 1 in the inner loop from statement `S4` to statements `S1` and `S2` on `d[i-1]`. 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.

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 and represent the *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:

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

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 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)

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

We present here the ILP formulation for the problem of minimizing $Cost(G)$.

$$\min \sum_{u \in V} c_u \tag{6}$$

$$w_e + r_v - r_u \geq 0, \quad \forall e = (u, v) \in E \tag{7}$$

$$c_u \geq w_e + r_v - r_u, \quad \forall e = (u, v) \in E \tag{8}$$

The objective function of our ILP formulation is given by the relation (6). The constraints (7) ensure that we have a legal retiming. The cost of a node after retiming is given by the inequation (8). 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 (6) ensures that the maximal value is reached by c_u, giving the expected cost for each node.

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.

Minimizing $Cost(G)$ can be solved in *polynomial time* $O(|V||E|.(\sum_{e \in E} w(e)))$ by using a the dual form of the problem to reduce it to a minimal cost flow problem (see [8, 16] for details).

5. Conclusion and Future Work

We have presented in this paper optimal algorithms to minimize the memory size needed for temporary arrays by loop fusion and to optimize the data locality within a loop nest by loop alignment. These transformations are independent from any architectural constraint or parameter.

Loop alignment can help to remove Fusion Preventing Edges from the original graph by modifying the dependence distance for some arrays and allow fusions that would have not been feasible without prior modifications. Loop fusion combined with loop alignment improves data locality within a loop nest and increase the search space and potential benefits of further optimizations such as cache level memory management and data layout organization.

As these transformations are source to source they can be easily integrated to an existing CAD tool-chain as a design preprocessor that can be run prior to any other tool of the chain. Therefore it will enable new optimizations obtained using existing memory and power management techniques in CAD tools.

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, F. Catthoor, J. Bormans, and H. D. Man. Code Transformations for Reduced Data Transfer and Storage in Low Power Realisation of mpeg-4 full-pel Motion Estimation. In *IEEE International Conference on Image Processing, ICIP*, volume 3, pages 985–989, Chicago, Oct. 1998.

[3] S. Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, Feb. 1993.

[4] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. *Surviving the SOC Revolution: A Guide to Platform-Based Design*. Kluwer Academic Publishers, 1999. ISBN 0–7923–8679–5.

[5] K. Danckaert. *Loop Transformations for Data Transfer and Storage Reduction on Multiprocessor Systems*. PhD thesis, Katholieke Universiteit Leuven – IMEC, May 2001.

[6] E. De Greef. *Storage Size Reduction for Multimedia Application*. PhD thesis, Katholieke Universiteit Leuven – IMEC, Jan. 1998.

[7] A. Fraboulet, K. Godary, and A. Mignotte. Loop Fusion for Memory Space Optimization. In *International Symposium on System Synthesis (ISSS'01)*. IEEE Computer Society Press, Oct. 2001.

[8] A. Fraboulet, G. Huard, and A. Mignotte. Loop Alignment for Memory Accesses Optimization. In *International Symposium on System Synthesis (ISSS'99)*, pages 71–77. IEEE Computer Society Press, Nov. 1999.

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

[10] C. Kulkarni. *Cache Optimization for Multimedia Applications*. PhD thesis, Katholieke Universiteit Leuven – IMEC, Feb. 2001.

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

[12] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

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

[14] P. R. Panda, N. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. In *International Symposium on System Synthesis*, 1996.

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

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