Laboratoire d'Ingénierie de l'Informatique Industrielle
Institut National des Sciences Appliquées de Lyon

Loop Fusion for Memory Space Optimization

Antoine FRABOULET
Karen GODARY
Anne MIGNOTTE

April 2001

Research Report N° 2001–04

Institut National des Sciences Appliquées de Lyon
69621 Villeurbanne Cedex, France
Téléphone : +33(0)4.72.43.83.83
Télécopieur : +33(0)4.72.43.85.00
Adresse électronique : l3i@insa-lyon.fr

Loop Fusion for Memory Space Optimization

Antoine FRABOULET
Karen GODARY
Anne MIGNOTTE

April 2001

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.

Keywords: Memory Optimization, Code Transformation, Codesign, Loop Fusion (Merging)

Résumé

Les systèmes portables ou embarqués, ainsi que les technologies dites submicroniques, ont rendu le critre de consommation crucial. La mémoire est connue pour être une source très importante de consommation. De plus, les applications multimédia font un usage très intensif de la mémoire. C'est pourquoi nous proposons de nouvelles techniques pour optimiser une description comportementale d'application multimédia avant le partitionement matériel-logiciel (*Codesign*). Ces transformations sont effectuées sur les boucles "for" qui constituent la partie principale manipulant les tableaux dans le code multimédia. Cet article présente un algorithme optimal pour réduire l'utilisation de tableaux temporaires par fusion de boucles. Bien que cet algorithme ne soit pas polynomial, les expérimentations ont montré qu'il est très efficace.

Mots-clés: optimisation mémoire, transformation de code, conception conjointe (codesign), fusion de boucles

Contents

List of Figures

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 Register Transfer Level, transformations are performed on the control state charts by splitting or merging nodes. At every stage of the design we can apply transformations used in software compilers [ASU86] [BGS94].

In this paper, we will focus on the design of data flow 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. In this paper, we want 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 [CCH+99] 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 [De 98] 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 [MCJM96] [PDN96] [PDN99]. 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 [ACFS94]. 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 loop transformations [Kul00] [MCT96].

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 [BGS94] [Wol95] so as to reduce the consumption in power due to memory by enabling powerful optimizations. Further optimizations are to be applied later in the design flow, when more architectural parameters are set for the design. However, the other steps and interactions between steps 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 by reducing the size of temporary arrays. These arrays are produced and used in specific loops and they are not used elsewhere in the code. An array written in a loop and read in another one must be stored in memory between these operations. Once the production and consumption of values stored in the array have been merged within the same loop body a scalar replacement [CK94] or in-place mapping [De 98] techniques can be used to reduce the size of memory needed by the computation.

Figure 1 shows an exemple of such fusion. On Figure 1(a) the loop `L1` produces values stored in array `A` and the loop `L2` consumes these values. If we merge these two loops then we obtain the code on Figure 1(b) where the array `A` has been replaced by a scalar `a` thus reducing the size of needed memory for the application.

```
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 of array `a` after loop fusion

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. 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 In-Place Mapping. Our `Memory Cost` function is defined as following:

$$\text{Memory Cost} = max \left\{ \sum_{v \in \text{Live variables}(t)} Size(v), \forall t \right\}$$

The loop fusion for maximal reuse proposed by McKinley and Kennedy [MK93] is a particular case of our problem and has been proven NP-hard. However, we propose a very efficient algorithm that is very fast in practice and solves our problem with an optimal solution. A survey on loop fusion complexity can be found in [Dar99].

2.1 Modeling the Problem

We use a *Data Flow Graph (DFG)* ($G = (V, E, A)$) representation to model the problem. 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 $a \in A$ has an associated weight $size(a)$.

We assume that the reader is familiar with concept of data dependence [BGS94]. A data dependence between two array references is represented by a hybrid distance/direction vector $\vec{\delta} = \{\delta_1 \dots \delta_2\}$ 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. 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 $a \in A$ 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 to nodes.

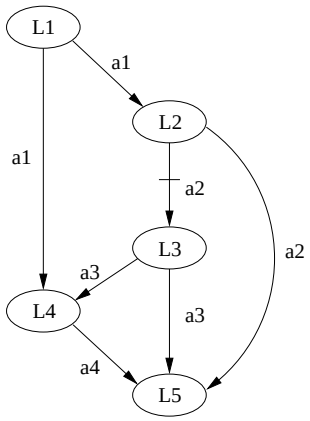
Figure 2(b) represents the dependence graph computed from the source code on Figure 2(a). Loops `L2` and `L3` cannot be fused due to the dependence carried by `a2`. If the loops were merged the code would read `a2[i+1]` (from `L3`) before its computation (from `L2`) within the same iteration.

Isolated statements are also considered as nodes. Dependences between code statements and loops are preserved as we perform code reorganisation 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.

```
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) dependence graph

Figure 2: Modeling dependences

2.2 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.

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 3.

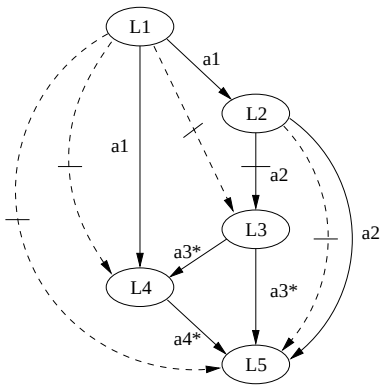


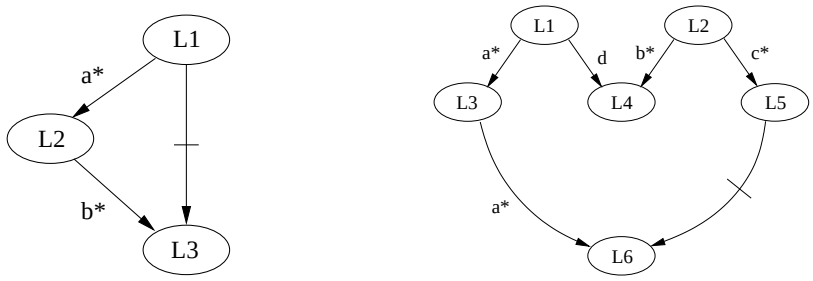
Figure 3: Transitive closure of the DFG for removable array detection

Array `a2` cannot be removed because there is a direct fusion preventing edge between loops `L2` and `L3`. The same situation occurs for array `a1` where there is a path between `L1` and `L4` that goes through an FPE. 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.

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 Conflicts 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 4(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 4(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 4: Conflict Between Potientialy Removable Array

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.

2.3.1 Conflicts 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 [GM84] 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 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 an 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] $\leftarrow$ true
  for all $t \in V$ do
  begin
    if $t = v$ and $t \neq u$ then
      $C \leftarrow C \cup \{v \in V | mark[v] = true\}$
    else if $e = (k, t) \in E$ and e is starred and mark[t] = false
      exploration(G,t,u,v)
    else if $e = (t, k) \in E$ and mark[t] = false
      exploration(G,t,u,v)
  end
  mark[k] $\leftarrow$ false
end

for all $e = (u, v) \in V$, **exploration**(C,u,u,v)

Figure 5: Graph exploration for problem detection between a node u and a node v

This algorithm is clearly not polynomial as it performs an exploration of the graph. However, this step is very efficient in practice and it has never been time consuming during our experiments (see Section 3.)

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 Resolution

In this section we present the ILP formulation [Sch86] for the problem of breaking all dependency cycles 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 removable array will be taken out of the set.

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 2.2.

We associate a binary variable x_a to each starred array a that could be removed but which has been included in a cycle during the previous step. If this variable is set to 0 then the array will be considered for removal by fusion otherwise the array will cease to be starred.

For each cycle $\mu \in C$ detected in section 2.3.1 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_a on a path must be greater than or equal to 1 in order to remove at least one array.

$$\min \sum_{a \in A} size_a * x_a \tag{1}$$

$$0 \leq x_a \leq 1, \forall a \in A \tag{2}$$

$$\sum_{a \in \mu^+(c)} x_a \geq 1, \forall \mu \in C \tag{3}$$

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

If the graph is a multigraph then we introduce extra variables $x_{a...b}$ that represent all the arrays carried by two or more dependences $u \stackrel{a*}{\rightarrow} v$ and $u \stackrel{b*}{\rightarrow} v$ between two nodes u and v of the graph. The variable $x_{a...b}$ replaces variables $x_a \ldots x_b$ in inequation 3 whenever the multi-edge appears in μ^+. A variable $x_{a...b}$ will be set less or equal to each variable $x_a \ldots x_b$ associated with the arrays of the multi-edge. If x_{ab} 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 can be set to 1 without interfering with other arrays on the multi-edge.

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 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 in which if there is a path $u \stackrel{*}{\rightarrow} v$, 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:

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

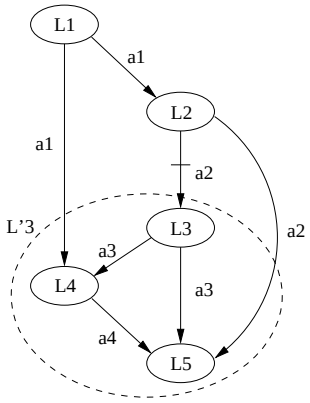
Figure 6 represents the modified dependence graph and the code corresponding to the loop fusion. Arrays `a3` and `a4` can be now replaced by scalars within the loop `L'3`.

3 Experimentations

We have implemented this algorithm and tested it using randomly generated graphs. We used the ILP solver `LP_SOLVE` freely avalable from its ftp site [Ber]. Generated graphs were ranging from 10 to 500 nodes with multiple dependencies and arrays. Although the cycle enumeration step is exponential in theory as well as the ILP resolution we did not notice any graph for which the complete algorithm took more than 0.1 second including cycle enumeration and ILP resolution. Tests were executed on a Pentium II 450MHz machine.

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 but is very efficient in



(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 6: Clustering and Code output

practice. Furthermore, real life problem size are small as the number of nodes in our graph represents the number of loop nests in a program.

As it is a source to source compiler it 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. 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 developped for memory accesses optimization [FHM99]. Loop alignement can help to remove FPE from the original graph and can also reduce the dependence distance for some arrays. Loop fusion combined with loop alignement improves data locality within a loop nest. Once a loop has been aligned, improved in-place mapping and scalar replacement results can be achieved over the data for both size and locality.

The algorithms and techniques presented here will be extended to deal with conditional execution ("if") in order to be able to model real-life applications. Therefore we want to use the *Program Dependence Graph* defined in [FOW87] by Ferrante et al. to take into account both data flow and control flow dependencies for source to source loop transformations.

Acknowledgments

We would like to acknowledge and thank Baptiste Jeudy and Guillaume Huard for their fruitful discussions and help on this work.

References

[ACFS94] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principes, Techniques, and Tools.* Addison-Wesley, 1986.

[Ber] Michel Berkelaar. Lp solve distribution. Ftp Archive, ftp://ftp.es.ele.tue.nl/pub/lp_solve/.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[CCH+99] 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.

[CK94] Steven Carr and Ken Kennedy. Scalar Replacement in the Presence of Conditional Control Flow. *Software Practice & Experience*, 2(1), January 1994.

[Dar99] Alain Darte. On the complexity of loop fusion. In *IEEE PACT*, pages 149–157, 1999.

[De 98] Eddy De Greef. *Storage Size Reduction for Multimedia Application.* Phd thesis, IMEC, January 1998.

[FHM99] Antoine Fraboulet, Guillaume Huard, and Anne Mignotte. Loop Alignment for Memory Accesses Optimization. In *International Symposium on System Synthesis (ISSS)*, pages 71–77, November 1999.

[FOW87] 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.

[GM84] M. Gondran and M. Minoux. *Graphs and Algorithms.* John Wiley, 1984.

[Kul00] Chidamber Kulkarni. *Cache Optimization for Multimedia Applications.* Phd thesis, IMEC, February 2000.

[MCJM96] 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)*, pages 20–25, La Jolla, California, November 1996.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[MK93] 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*, pages 301–320. Lecture Notes in Computer Science 768, Springer-Verlag, 1993.

[PDN96] 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.

[PDN99] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-On-Chip.* Kluwer Academic Publishers, 1999. ISBN 0–7923–8362–1.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1986.

[Wol95] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.