

ACKNOWLEDGMENT

The authors would like to thank three anonymous referees for their valuable comments and suggestions.

REFERENCES

- [1] E. J. Hannan, "The estimation of frequency," *J. Applied Probability*, vol. 10, pp. 510-519, 1973.
- [2] D. Kundu, "Asymptotic theory of least squares estimator of a particular nonlinear regression model," *Statist. Probability Lett.*, vol. 18, pp. 13-17, 1993.
- [3] —, "Consistency of the undamped exponential model for a restricted parameter space," *Commun. Statist. Theory Methods*, vol. 24, no. 1, 1995.
- [4] D. Kundu and A. Mitra, "A note on the consistency of undamped exponential signals model," *Statist.*, vol. 26, 1996.
- [5] A. Mitra and D. Kundu, "Asymptotic behavior of least squares estimates of superimposed exponential signals," *P. C. Mahalanobis Birth Centenary Volume*, A. K. De, Ed. Indian Statistical Institute, 1993, pp. 653-663.
- [6] C. R. Rao, L. Zhao, and B. Zhou, "Maximum likelihood estimation of 2-D superimposed exponential signals," *IEEE Trans. Signal Processing*, vol. 42, no. 7, 1994.
- [7] C. R. Rao and L. C. Zhao, "Asymptotic behavior of maximum likelihood estimates of superimposed exponential signals," *IEEE Trans. Signal Processing*, vol. 41, pp. 1461-1463, 1993.

Analysis of Multidimensional DSP Specifications

Ingrid Verbauwhede, Chris Scheers, and Jan M. Rabaey

Abstract—Data flow languages are a natural and more formal way to describe the flow of computations in a DSP application. However, when the language contains extended array constructs, extra data dependency checks are needed. This correspondence describes a new model to represent multidimensional arrays and presents several data flow analysis techniques for multidimensional arrays. Results show very fast running times (<1 s) for problems of more than 100 nodes.

I. INTRODUCTION

Data flow languages are a convenient way to capture the flow of computations in a DSP application. It can be described with freedom of memory side effects, and the inherent parallelism in the algorithm is explicitly expressed. To ensure locality of effect, data flow languages adhere to the single assignment property [1]. This means that a variable may appear on the left side of an assignment only once within the area of the program in which it is active.

If a data flow language is selected, it requires different data dependency checks from a compiler, and it has different consequences for the memory management compared with a procedural language. In procedural languages, such as C or Fortran, arrays are considered

Manuscript received February 3, 1995; revised February 2, 1996. The associate editor coordinating the review of this paper and approving it for publication was Dr. Keshab K. Parhi.

I. Verbauwhede and C. Scheers are with TCSI, 2121 Allston Way, Berkeley CA 94704-1301 USA (e-mail: ingrid@tcsi.com).

J. M. Rabaey was with TCSI, 2121 Allston Way, Berkeley CA 94704-1301 USA. He is now with the University of California at Berkeley, Berkeley, CA, 94720 USA.

Publisher Item Identifier S 1053-587X(96)09063-0.

objects in memory that are being modified. This mechanism makes it nearly impossible to avoid side effects in a procedural description. Side effects are, e.g., modifications of data in a memory location between a read and a write, or out-of-bounds memory accesses. Making the data consistent in the memory is the responsibility of the programmer. For the compiler, the operations need to be executed in the order as specified in the input description; hence, they are called "procedural," "sequential," or "imperative" languages [1]. In a data flow language, arrays in the language are decoupled from memory locations. It is the compiler that derives the lifetime of the signals in the array, the memory allocation, memory assignment, etc. If the language contains constructs to address individual signals in the array or parts of the array by means of index expressions, this requires more complex data dependency checks.

This fundamental difference has consequences for the data dependency checks. A procedural language has true, anti, and output dependencies. A data flow language has a single assignment check, production-consumption check, data dependency check, and a loop instance check. This is the topic of this correspondence.

For scalar signals, single assignment checking and creating data dependencies between operations is based on their name. Each scalar signal has its own name, and when ordering the statements, a definition of a scalar has to occur before a consumption. To address this problem for arrays, they need to be considered in exactly the same way as scalars. For arrays as well, an applicative assignment creates a unique name to value binding, which is different from a procedural language where the name to value binding changes.

In this correspondence, the SILAGE language is used to illustrate the data dependency checks [8]. In SILAGE, arrays are considered a collection of signals upon which the same or similar operations are performed. The applicative assignment requires a unique name for each signal in the array. The usage of loops and arrays is, thus, a convenient short-hand notation to indicate that operations are repeated on an array of signals.

A. Functional Languages and Data Dependency Checks

Most functional languages have been designed to expose maximum parallelism such that the execution of application programs on parallel machines is improved. Examples are Id [4], EPL [15], SISAL [5], [6], CRYSTAL [3], and LUCID [14]. Each of these languages contains data structures to group values into arrays, streams, or lists. Depending on the type of data structures and the flexibility of creating and using values in the arrays or lists, different data flow analysis checks are required from the compiler. Functional languages are also used for regular array synthesis to expose regularity [2], [11].

Id [4] contains two data types to describe a collection of values: arrays and lists. Arrays in Id can be built by regions. Each region, which corresponds to part of the array, is defined by a different function. To avoid single assignment problems, the programmer has to write the description in such a way that the regions are disjoint, while a run time check will flag any errors. In EPL, if two assignments try to create the same variable, the single assignment is enforced by generating object code that only executes the first assignment. The second assignment, which tries to reassign the variable, is ignored [15]. SISAL [5], [6] is an applicative language that contains 1-D arrays. Multidimensional arrays are created as arrays of arrays. Arrays are created by a specific operator that binds the i th element of an array to the i th iteration [6]. If an array element is a function of a previous value of the array, only the values of the previous

iteration can be accessed (with the "old" operator) [6]. This simplifies the single assignment and the production consumption check and makes a general single assignment check redundant [5]. LUCID contains (multidimensional) lists to group values [14]. Since it is an "intentional" language, accesses are referred to in a relative way by using intensional operators such as next, previous, successor etc.

For describing real-time signal processing applications, graphical data flow representations are also being used. Examples are the SIGNAL language and its graphical block diagram representation [10] and the graphical data flow representation used by PTOLEMY [9]. The synchronous data flow model of PTOLEMY has been extended to multidimensional streams [9]. A data flow graph consists of a network of nodes, called actors, which are connected by arcs, carrying the data. To each actor, two numbers are assigned that describe the number of tokens being produced and consumed by the actor on each firing. To model multidimensional streams, tuples of tokens are assigned, which represent the number of tokens in each dimension. Tokens are produced or consumed in a block (all elements at the "same time"). If individual elements of a block are needed, the designer has to go to a smaller granularity, e.g., by using a downsampling actor. Because of this block approach, a single assignment check is redundant. A production consumption check is replaced by solving the balance equations, which describe the conditions under which the data flow diagram is executable.

SILAGE allows different assignments to define parts of an array. It also allows complex index expressions. As a result, a single assignment check is needed. Second, individual signals in an array can be accessed at random. Therefore, a production consumption check is needed. Third, checks are needed to create data dependencies between productions and consumptions of signals and to check if these dependencies occur in the same or different loop instances. Each of these checks is described in this correspondence.

In Section II, the representation of arrays is introduced. In Section III, the different data dependency checks are discussed. In Section IV, some results and discussion are presented. Section V contains the conclusion.

II. REPRESENTATION OF ARRAYS

A data flow graph is commonly used to represent the statements in a data flow description [1]. There is a one-to-one correspondence between a data flow description and a data flow graph. Nodes represent operations, and edges represent data dependencies between operations. To model arrays of signals, *production* and *consumption nodes* are introduced.

Definition: An occurrence of an indexed array at the left-hand side or at the right-hand side of a statement is called a production or a consumption node, respectively.

This is illustrated on Fig. 1. Because of the single assignment property, scalar signals have a unique name, and there will be only one edge in the flow graph with this name, e.g., in Fig. 1(a), there is only one data edge with the name "b." For arrays, however, several data edges can have the same array name but with different index ranges. Different statements can define different disjoint parts of the array or can consume different parts of an array. Each production statement therefore ends in a production node, and each consumption starts from a consumption node. Data dependency analysis will determine the dependencies between the production and the consumption nodes. This is illustrated in Fig. 1(b).

Definition: To define exactly the index range of a production/consumption node, a set of linear equalities and inequalities is associated to the node. This set of equations defines the subscripts

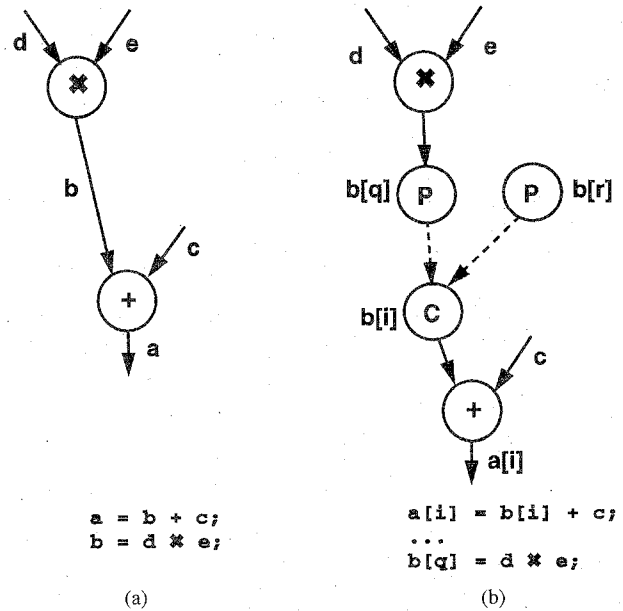


Fig. 1. Production and consumption nodes for arrays of signals.

of the node as a function of the loop indices and the time index¹

$$\bar{s} = A \times \bar{i} + \bar{c} \quad \text{with} \quad B \times \bar{i} + \bar{d} \geq 0. \quad (1)$$

\bar{s} is the subscript vector, and \bar{i} is the loop iterator vector that includes the time index. A, B, \bar{c} , and \bar{d} are matrices (capital characters) and vectors (overlined small characters) of integer values. Individual loop iterators, which are scalar variables, are denoted with small letters.

The equations, matrices, and vectors used in this text can also contain subscripts C and P to refer to the index values during consumption or production. Subscripts are numbered s_1 or s_2 , referring to the first or second subscript, etc. t_P and t_C refer to the time loop index and $i_{C1}, i_{C2}, i_{P1}, i_{P2}$, etc. to general loop indices.

III. DATA DEPENDENCY CHECKS FOR ARRAYS

If the single assignment language allows the definition of an array by more than one assignment, the compiler needs to check if these definitions do not overlap. This is the *single assignment check*. Second, the same flexibility applies to the consumption of arrays. Several statements can use parts of the array. Therefore, the compiler has to check that for each signal being used, there are corresponding production statements. This is called the *production consumption check*.

The data dependencies between productions and consumptions cannot only be based on the name. This would introduce false and circular dependencies. A more accurate check is needed to create the data dependencies based on the index expressions. This is called the *data dependency check*.

A data dependency check alone is not enough to determine the sequence of execution for scheduling or topological ordering. It can result in circular graphs. Edges that cross loop iterations should not be taken into account for topological ordering. The fourth check that annotates this information on the dependency edges between production and consumption nodes is called the *loop instance check*.

¹Expressions in SILAGE denote infinite streams of values. The time loop is considered the most outer loop surrounding an application program. Time runs from 0 to ∞ . Hence, this index has no upper bound.

```

(i : 0 .. 10) :: (j : 0 .. 10) ::
begin
  a[i + j] = . . . ;
end;

```

Fig. 2. Example of a production node that needs to be compared to itself.

A. Single Assignment Check

If several production nodes define parts of the same array, one has to check if those definitions do not overlap. This is equivalent to solving the following problem.

Problem Definition: Is there an integer solution for the subscripts subject to the following set of linear equalities and inequalities, which describes the intersection of two production nodes?

$$\begin{aligned} \bar{s} &= A_{P1} \times \bar{i}_{P1} = A_{P2} \times \bar{i}_{P2} \\ B_{P1} \times \bar{i}_{P1} + \bar{d}_{P1} &\geq 0, \quad B_{P2} \times \bar{i}_{P2} + \bar{d}_{P2} \geq 0 \end{aligned} \quad (2)$$

with \bar{s} = vector of subscripts, and $\bar{i}_{P1}, \bar{i}_{P2}$ = vector of loop iterators surrounding production node 1 and node 2, respectively. If a solution exists to this ILP problem, single assignment is violated. This comparison in pairs is performed between all production nodes of the same array.

Each production node has to be compared not only with the other production nodes but also with itself. Index expressions can be such that several index expressions define the same value. This is illustrated in Fig. 2; the signal [2] is defined three times by the index tuples: (0, 2), (1, 1) and (2, 0).

Problem Definition: Is there an integer solution to the following ILP problem? It describes the intersection of a production node and itself. An extra constraint is added that requires the index vectors to be different.

$$\begin{aligned} \bar{s} &= A_{P1} \times \bar{i}_{P1} = A_{P2} \times \bar{i}_{P2} \\ B_{P1} \times \bar{i}_{P1} + \bar{d}_{P1} &\geq 0, \quad B_{P2} \times \bar{i}_{P2} + \bar{d}_{P2} \geq 0 \\ \bar{i}_{P1} &\neq \bar{i}_{P2}. \end{aligned} \quad (3)$$

Because a "not equal" constraint cannot directly be added to an ILP problem, two ILP problems are solved, where the $\bar{i}_{P1} \neq \bar{i}_{P2}$ constraint is replaced by two inequality constraints: $\bar{i}_{P1} \geq \bar{i}_{P2} + 1$ and $\bar{i}_{P2} \geq \bar{i}_{P1} + 1$. If one of the problems has a solution, single assignment is violated.

B. Production-Consumption Check

Each integer point in a consumption node needs to be covered by one or more production nodes. Simply taking the intersection of the production and the consumption nodes does not solve the problem. We need to know whether integer points of the consumption node lie outside the production nodes. Therefore, we need to check the intersection of the consumption node with the inverse of the production node.

Problem Definition: Taking the intersection of two nodes corresponds to solving one system of equations, which is the total of the constraints of both nodes, i.e., "anding" the constraints, as is indicated in (4). In (4) and (5), con stands for constraint.

$$\begin{aligned} &\text{node 1} \cap \text{node 2} \\ &\Leftrightarrow \\ &(\text{con } 1_1 \wedge \text{con } 2_1 \wedge \cdots \wedge \text{con } N_1) \\ &\quad \wedge (\text{con } 1_2 \wedge \text{con } 2_2 \wedge \cdots \wedge \text{con } M_2). \end{aligned} \quad (4)$$

Checking the intersection of a node—in this case, a consumption node—and the inverse of a second node—in this case, a production

```

CheckProductionsConsumptions(Graph)
for each array,
  for each ConsNode
    IntersectWithComplement (ConsNode, ProdNodeList);

```

```

IntersectWithComplement(ConsNode, ProdNodeList)
if (ProdNodeList empty)
  ERROR: consumption without production;
else
  take first Prodnode from the list;
  for each constraint of ProdNode
    Problem = ConsNode + inverted constraint;
    Solution = ILPSolve(Problem);
    if (Solution)
      IntersectWithComplement (Solution, ProdNodeList->Next);
  else
    Problem = ConsNode;
    continue;
endfor;
return OK;

```

Fig. 3. Pseudo code for the production consumption check.

node—results in several systems of equations. Each system corresponds to the consumption node augmented with one of the inequality constraints of the production node inverted. This is indicated by the "or" operation in (5).

$$\begin{aligned} &\overline{\text{node 1}} \cap \text{node 2} \\ &\Leftrightarrow \\ &\overline{(\text{con } 1_1 \wedge \text{con } 2_1 \wedge \cdots \wedge \text{con } N_1)} \\ &\quad \wedge (\text{con } 1_2 \wedge \text{con } 2_2 \wedge \cdots \wedge \text{con } M_2) \\ &\Leftrightarrow \\ &(\overline{\text{con } 1_1} \wedge \text{con } 1_2 \wedge \text{con } 2_2 \wedge \cdots \wedge \text{con } M_2) \vee \cdots \vee \\ &(\overline{\text{con } N_1} \wedge \text{con } 1_2 \wedge \text{con } 2_2 \wedge \cdots \wedge \text{con } M_2). \end{aligned} \quad (5)$$

Solution Approach: If one (or more) of the systems has a solution, it describes the part(s) of the consumption node, which lies outside the production node. This new system has to be covered by the remaining production nodes. This process is repeated recursively until the node is covered or until no production nodes are left. The latter indicates a violation. The production and consumption nodes are represented as follows:²

$$A_P \times \bar{s} + \bar{d}_P \geq 0 \quad \text{and} \quad A_C \times \bar{s} + \bar{d}_C \geq 0.$$

The i th equation to describe the production node is of the form

$$\bar{a}_{iP}^T \times \bar{s} + d_{iP} \geq 0.$$

Inverting an inequality constraint in a system of integer linear inequalities consists of inverting the sign of each component and subtracting one (to obtain, again, a "greater than or equal" equation). This results in

$$-\bar{a}_{iP}^T \times \bar{s} - d_{iP} - 1 \geq 0.$$

This equation is added to the consumption node description. If this ILP problem has a solution, the next production node is taken from the list, and its equations are inverted one by one. The recursive algorithm for this production-consumption check is given in Fig. 3 and illustrated in Fig. 4.

²A subscript notation is used, i.e., all loop iterators and redundant equations have been removed.

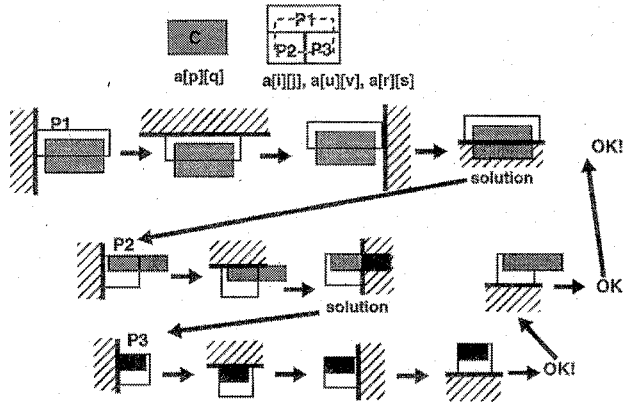


Fig. 4. Recursive production-consumption check.

For efficiency reasons, one restriction is applied to this check: The multiplication constants in the description of the production node have to be 1, -1 , or 0. It means that the entries in the A_P matrix of production node $A_P \times \bar{s} + \bar{d}_P \geq 0$ are 1, -1 , or 0. The reason is that a production node with A_P matrix entries different from 1, -1 , or 0 does not cover all integer points between the inequality boundaries. As such, a production node is a collection of smaller nodes. This is illustrated in Fig. 5. For the first subscript, the matrix A_P has entries 2 and -2 . It describes five sets of integer points: one for s_1 equal to 2, 4, 6, 8, and 10, respectively. Inverting the inequality constraints degenerates into inverting the inequalities of all the subnodes. The proposed technique can still be applied but can become inefficient if the number of nodes grows large. To solve this problem, algebraic techniques that group the subnodes into larger nodes need to be applied.

C. Creation of Data Dependencies

The dependencies between productions and consumptions are created by checking the intersection between a production and a consumption node, which is similar to the single assignment check of (2).

Problem Definition: Does there exist an integer solution for the subscript vector \bar{s} for the following set of linear equalities and inequalities, which describes the intersection of a production node and a consumption node?

$$\begin{aligned} \bar{s} &= A_P \times \bar{i}_P = A_C \times \bar{i}_C \\ B_P \times \bar{i}_P + \bar{d}_P &\geq 0, B_C \times \bar{i}_C + \bar{d}_C \geq 0. \end{aligned} \quad (6)$$

If there is a solution, a dependency exists between the production and the consumption node. For each array, a comparison in pairs between each production node and each consumption node is needed.

This dependency information only describes pure data dependencies. It is used in our compiler to perform topological ordering between groups of statements that reside in different loop bodies. If they are independent, they can be executed in parallel. Data-dependency information is useful but not sufficient for determining the order of operations. Therefore, control flow information needs to be added.

D. Loop Instance Check

A legal ordering of the operations, which makes the code executable, is obtained if productions occur before consumptions. Techniques such as topological ordering or scheduling will perform this task. However, in general, a production or consumption node represents a set of signals. Not all signals in this set have to be

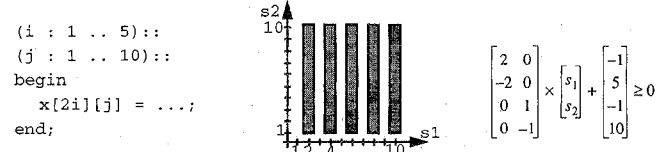


Fig. 5. Production node with corresponding subscript space and set of subscript equations.

produced before the first of the consumptions can start. Requiring that all productions occur before the consumptions can result in cyclic dependency requirements. To break these cycles in the dependency graphs, a notion of ordering *within* the sets of signals, which is control flow information, has to be imposed onto this data dependency checking step.

Definition: Control flow is defined as all possible execution paths [20]. For the execution sequence of the loops, a "standard execution order" in lexicographical order, is assumed [20]

Statement instance $S(i, j, k)$ is executed before $S(i', j', k')$ if and only if $(i, j, k) < (i', j', k')$

where $<$ denotes lexicographic ordering in the set of iteration vectors. No assumptions are made on the execution sequence of nodes within the body of a loop.

Problem Definition: To check if an edge needs to be considered within a loop body, the following ILP problem is solved:

$$\begin{aligned} \bar{s} &= A_P \times \bar{i}_P = A_C \times \bar{i}_C \\ B_P \times \bar{i}_P + \bar{d}_P &\geq 0, B_C \times \bar{i}_C + \bar{d}_C \geq 0 \\ \bar{i}_P &= \bar{i}_C. \end{aligned} \quad (7)$$

The first three equations describe the constraint for a data dependency. The fourth equation requires that the common loop iterators be equal for the production and consumption node. \bar{i}_P and \bar{i}_C refer to the common loop iteration vectors. If the above problem has a solution, it means that production and consumption occur in the same instance of loop iterators, and the edge needs to be considered during topological ordering or scheduling.

E. Solution Approach

Previous SILAGE parsers, such as the S2C simulator used in [16], make use of symbolic enumeration techniques. It consists of creating a table of boolean values for each array. Entries in the table indicate individual signals that have been created. Next, the algorithm is symbolically executed, which mimics a run-time execution. During this execution, left-hand side productions are marked as being produced, and for right-hand side consumptions, the table is checked if the value has been produced. The symbolic enumeration technique is robust and can handle even nonlinear index expressions such as $a[i * j]$. It has the disadvantage that the tables grow with the sizes of the arrays since each signal has its entry.

The techniques proposed here are based on an ILP formulation. Compared with a symbolic enumeration approach, this technique is very efficient. However, it puts some constraints on the expressions for the nodes. For an ILP approach to be applicable, the equations need to be linear expressions [20]. Each problem is solved by the OMEGA ILP solver [12]. These ILP problems do not depend on the size of the array but depend on the number of constraints to describe a node or an intersection of nodes. The number of constraints depends on the dimensions of the array and the number of loops surrounding the node, which is, in most practical cases, very small. The OMEGA test makes use of the Fourier-Motzkin elimination method to solve a system of linear inequalities [13]. In the worst

V. CONCLUSION

When a data flow language is selected to describe DSP applications, different data dependency checks are needed. This is especially the case if the language contains array data structures with complex index expressions. In this correspondence, a representation model for arrays in a data flow graph is presented. This model is used for several data dependency checks: a single assignment check, a production consumption check, a data-dependency check, and a loop instance check. Each problem is formulated as an ILP problem and is solved with the Omega test. Tests on large examples have shown run times of less than 1 s for applications consisting of more than 150 production and consumption nodes.

REFERENCES

- [1] W. B. Ackerman, "Data flow languages," *IEEE Comput.*, pp. 15–24, Feb. 1982.
- [2] G. Bu, "Systolic array implementation of nested loop programs," in *Proc. Int. Conf. Application Specific Array Processors*, Princeton, NJ, Sept. 1990, pp. 31–42.
- [3] M. Chen, Y. Choo, and J. Li, "Crystal: Theory and pragmatics of generating efficient parallel code," in *Parallel Functional Languages and Compilers*, B. Szymanski, Ed. New York: ACM, 1991, ch. 7.
- [4] K. Ekanadhan, "A perspective on Id," in *Parallel Functional Languages and Compilers*, B. Szymanski, Ed. New York: ACM, 1991.
- [5] J. Feo, D. Cann, and R. Oldehoeft, "A report on the Sisal language project," *J. Parallel Distributed Comput.*, vol. 10, pp. 349–366, 1990.
- [6] J. Feo, "Arrays in Sisal," *Arrays, Functional Languages and Parallel Systems*, L. Mullin, M. Jenkins, G. Hains, R. Bernecky, and G. Gao Eds. Boston: Kluwer, 1991, ch. 5.
- [7] G. Golub and C. Van Loan, *Matrix Computation*. Baltimore: John Hopkins Univ. Press, 1983.
- [8] P. Hilfinger and J. Rabaey, "DSP specification using the SILAGE language," in *Anatomy of a Silicon Compiler*, R. W. Brodersen, Ed. Boston: Kluwer, 1992, ch. 15.
- [9] E. A. Lee, "Multidimensional streams rooted in dataflow," in *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (A-23)*, M. Cosnard, K. Ebcioğlu, and J.-L. Gaudiot, Eds. Amsterdam: Elsevier (North-Holland), 1993.
- [10] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with signal," in *Proc. IEEE*, vol. 79, no. 9, Sept. 1991.
- [11] H. Le Verge, C. Mauras, and P. Quinton, "The alpha language and its use for the design of systolic arrays," *J. VLSI Signal Processing*, vol. 3, pp. 173–182, 1991.
- [12] W. Pugh, "The Omega test: A fast and practical integer programming algorithm for dependency analysis," *Commun. ACM*, vol. 8, pp. 102–114, Aug. 1992.
- [13] A. Schrijver, *Theory of Linear and Integer Programming*. New York: Wiley, 1986.
- [14] D. Skillikorn, "Stream languages and data-flow," in *Advanced Topics in Data-Flow Computing*, J.-L. Gaudiot and L. Bic, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1991, ch. 16.
- [15] B. Szymanski, "EPL—Parallel programming with recurrent equations," in *Parallel Functional Languages and Compilers*, B. Szymanski, Ed. New York: ACM, 1991, ch. 3.
- [16] J. Vanhoof, I. Bolsens, and H. De Man, "Compiling multi-dimensional data streams into distributed DSP ASIC memory," in *Proc. ICCAD'91*, Nov. 1991.
- [17] M. van Swaaij, F. Franssen, F. Catthoor, and H. De Man, "Modeling data flow and control flow for high level memory management," in *Proc. EDAC-92*, Feb. 1992, pp. 8–13.
- [18] I. Verbauwheide, C. Scheers, and J. Rabaey, "Memory estimation for high level synthesis," in *Proc. Design Automat. Conf., DAC-94*, San Diego, CA, June 1994, pp. 143–148.
- [19] W. Verhaegh *et al.*, "Modeling periodicity by Phideo streams," in *Proc. Workshop High Level Synthesis*, Dana Point Resort, CA, Nov. 1992.
- [20] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. New York: Addison-Wesley, ACM Frontier Series, 1990.