Array Mapping in Behavioral Synthesis

Herman Schmit and Donald E. Thomas
Dept. of ECE, Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

This paper discusses the mapping of arrays in a behavior to memories in an implementation. We introduce a design representation based on a variety of array grouping techniques and the binding of array groups to memory components with different dimensions, access times, and number of ports. The results of design actions are computed in terms of the number of memory components and the length of schedules in the behavior. We demonstrate the ability of a synthesis tool using this representation to generate designs that span the entire range of the memory design space.

1.0 Introduction

The goal of behavioral synthesis is to automatically create, from a behavioral specification, a custom hardware implementation that meets the design goals of the user. For many applications, the design of the memory architecture is critical in determining the cost and performance of the implementation. The most commonly used data structure in declarative specification languages is the array, which is assumed here to have a single index. In this paper we discuss the problem of mapping arrays, which have been specified in the behavior, to physical memories in an implementation, with the objective of satisfying the cost and performance goals of the user.

Both arrays and memory components are two-dimensional arrangements of bits. One dimension is the number of bits in a single array element, which is called the word width and shown horizontally. The other dimension is the number of elements in the array, which is called the array depth and shown vertically. Our design representation is based on the fact that arrays and memories can be tiled together in various ways to form array and memory groups. Array groups can be mapped to memory groups, as shown in Figure 1. A legal mapping is one in which every live array data bit is mapped to a distinct memory bit.

Memory cost is reduced by using memory components that have a lower cost per bit and by creating array groups that use most or all of the memory available in the allocated components. However, less expensive memories are usually slower, and the creation of array groups can create resource conflicts. The selection of an array-memory mapping therefore affects the way a behavior can be scheduled, and conversely, a schedule determines the array-memory mappings that are possible. For a design tool to make accurate cost and performance trade-offs it must deal with the coupled problems of scheduling and array-memory mapping simultaneously. In our approach we simultaneously determine the array-memory mapping and the array access schedule so that the interdependencies between these tasks are taken into account.

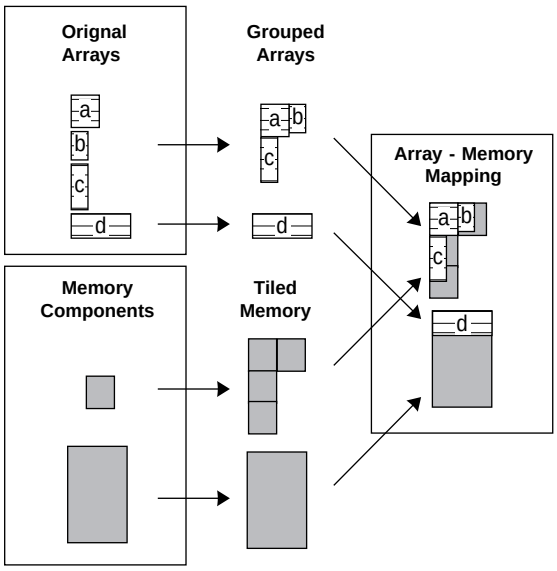

Figure 1. Array-memory mapping

The problem of synthesizing memory from array specifications has received significant attention recently. Ramachandran et al. [10] introduced the concept of array grouping, but their grouping techniques are limited, and rather than constructing the memory system out of components with specified costs, dimensions, access times, and ports, they determine the required size and number of ports of each memory needed to meet a specified schedule length and then use a model to determine area and access time. Karchmer and Rose [4] map arrays to memory components, but the user must specify the required access time for each array rather than a more abstract specification of performance such as the schedule lengths of schedule sets

in the behavior. The DSP synthesis algorithms described in [15] deal with the mapping of data streams (rather than arrays) to memories, and utilize the regularity of DSP access patterns to improve memory utilization. Unfortunately, not all applications can be easily described using streams, nor do all applications have the regularity of access of DSP applications. We attempt to deal with these sorts of applications.

In Section 2.0 we discuss the methods of grouping arrays. In Section 3.0 we discuss how we model the cost and performance of implementing an array group using a specified memory component. In Section 4.0, we demonstrate how array grouping, memory component binding and scheduling techniques are integrated into a simulated annealing based design tool. Finally, we demonstrate the use of this design tool by synthesizing implementations that span the entire memory cost and performance design space for a Viterbi search and a fuzzy control application.

2.0 Array grouping

We have identified three fundamental ways to group arrays:

- By *horizontal concatenation*, which is mapping data bits of each array to different data bits of the grouped array.
- By *vertical concatenation*, which is mapping data words of each array to different data words of the grouped array.
- By *time multiplexing*, which is mapping arrays with different lifetimes to the same address and data word space.

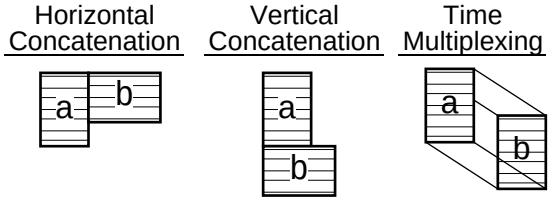

Figure 2. Array grouping operations

Horizontal concatenation creates wider grouped arrays and a shared address space. This shared address space reduces the number of address wires required to control the memory for these arrays. Also because of the shared address space, accesses to horizontally concatenated arrays must either be scheduled at different times or on different address ports (assuming there is more than one port on the memory component used). An exception to this rule is when the accesses are to the same index in each array. In that case both accesses can be performed at the same time on one address port, which can significantly improve memory bandwidth without requiring additional address busses or ports. This case occurs fairly frequently, because many behaviors contain parallel arrays.

If a single array within a horizontally concatenated array group is individually written, we have to assure that the data in other arrays in the group are not corrupted. In the worst case, this requires a read of the array group so that the original data can be written back into the other arrays. This lengthens effective write access time and requires significant control and data alignment logic, and is therefore rarely worthwhile.

Vertical concatenation creates deeper grouped arrays and a shared data word space, and is the only grouping operation discussed in [4] and [10]. The performance impacts of vertical concatenations are clear: the number of accesses to vertically concatenated arrays that may be scheduled simultaneously is limited to the number of ports on the memory component used to implement the array group.

Multiple options are available for generating non-intersecting address spaces for vertically concatenated arrays. These options include the use of address offsets and a variety of bit shuffling and inversion techniques that can be computed much faster and with substantial less hardware [11]. The address generation time required at each vertical concatenation is computed and used in the optimization of the memory system.

Time multiplexing is used to map arrays that have non-intersecting lifetimes into the same address and data word space. The width and depth of a time-multiplexed array group is the maximum of the width and depth of the grouped arrays. Lifetime information for arrays is determined from the lexical scope of the array declaration in the behavioral specification, although it would be more useful to use data flow analysis techniques [7][9].

Larger array groups can be created by recursive application of array grouping operations. For example in Figure 3, the horizontal concatenation of arrays **a** and **b** is vertically concatenated with array **c**. We represent such array groups with a binary tree: the leaves of the tree are the arrays in the behavior, and the other tree nodes are array grouping operations. This representation is similar to slicing structures using in floorplanning applications[16].

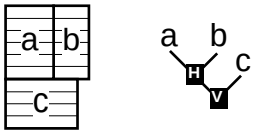

Figure 3. Combining array groupings

3.0 Memory component binding

As shown in Figure 1, array groups must be mapped to a set of one or more memory components that are

arranged so that every data bit in the array group has a distinct bit in the memory. We have made the simplifying assumption that each array group may be mapped to a set of only one type of component. For example, we cannot tile 1Mb and 4Mb DRAMs together to implement one array group, although we can use them in the same design for different array groups. Using this assumption, we can assign one type of component from our library to an array group, then determine the number of instances of that component necessary to cover the array group as well as the effective access time of the memory. This section describes the models we use to estimate the memory cost and access time given an array group and a component assignment.

3.1 Cost modeling

The simplest way to estimate the number of memory components necessary to implement an array group is to draw a bounding box around the array group and determine the number of components required to implement a memory space equal to that bounding box. This approach penalizes array groups that are not rectangular, as the example in Figure 4 shows.

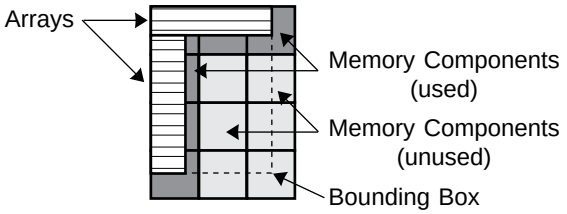

Figure 4. Counting memory components

To more accurately evaluate the cost of such an array grouping, we determine which memory components in the bounding box are actually used. The execution time of this procedure grows linearly with the number of memory components required for the bounding box, and is therefore undesirable when the size of a memory component is very small compared to the size of the bounding box. We have provided a user switch to determine whether bounding box or component counting technique during design optimization.

3.2 Performance modeling

The specified access time of a memory component is only part of the effective access time of a memory built out of that component. Other components of the access time include address computation, data and address bus propagation time and chip select decoding. Address bus and data bus propagation time usually grows linearly with the number of components on each bus. Decoding address bits into chip selects for a set of components usually increases with the log of the number of chip selects generated. We have implemented simple models based on these growth rates and basic glue logic speed so that the effective access time can be estimated for scheduling.

4.0 Automated synthesis

We have claimed that our representation allows for the exploration of a large cost-performance trade-offs. To show this, we have developed a general specification of system cost and used a general optimization procedure, simulated annealing [6], to explore this design space. It may be possible to reduce the generality of the system cost specification and use heuristics to find suitable solutions, but we find the two hour typical run time for the annealing algorithm is well worth the flexibility it affords.

Our specification of system cost specification uses three types of design metrics:

- the number of memory components of each type specified in the library (e.g. three 32K RAMs, and one 8K SRAM),
- the number of address, data and control wires necessary to connect with the memory components of each type, and
- the schedule length of each of the scheduling sets in the behavior.

The number of memory components can be used to construct an expression of total memory cost for the system. Similarly, the schedule lengths can be used to construct an expression of system performance. Reducing the number of address, data, and control wires saves physical pins on multiple chip designs and may improve the routability of on-chip memory designs.

If there are j types of components in the library and k schedule sets in the behavior, there will be $2j+k=m$ total metrics in the design. These m metrics, $(x_1, x_2, \ldots, x_m)$, are present in the system cost function, which is defined by the following equation.

$$\sum_{i=1}^{m} A_i x_i + \sum_{i=1}^{m} B_i \cdot max(x_i - C_i, 0) \qquad \text{(EQ 1)}$$

The user must specify the set of weights $(A_1, A_2, \ldots, A_m)$, to indicate the relative cost of the metrics. The user may also specify any number of constraints, C_i, and penalty weights, B_i (with $B_i >> A_i$) to express a limit on the value of a metric.

Our memory design optimizer uses a reconfigurable simulated annealing library [8], which implements a variety of cooling schedules and move selection procedures. We next describe the move set for this optimizer. The move set consists of basic moves, which are implied by

the design representation itself, and compound moves, which are sets of related basic moves that improve the annealing.

4.1 Basic moves

The first of the basic moves are horizontal concatenation, vertical concatenation, or time multiplexing of two arrays or array groups. Ungrouping of two arrays groups is also a basic move.

The next basic moves is the binding of a type of memory component to an array group. When an array group is bound to a memory component, the techniques described in Section 3.0 are used to determine the required number of components and effective access time of the arrays group.

The last basic move is the scheduling operation, which schedules an access in a specified step. Not all steps are legal for a particular access. Specifically, the address or the data for that access may not be computed yet, a situation we call *schedule overlap*, or the memory unit may be fully utilized by other accesses in that step, a situation we call *port overlap*. The annealer can schedule accesses illegally, but the total system cost is highly penalized for each step of schedule overlap or port overlap in the design.

Our annealer only schedules array accesses. Other operations are conjecturally scheduled using ASAP heuristics and functional unit constraints. After the array-memory mapping and array access schedule is determined, the results are given to SAM[2] to perform scheduling and hardware allocation for the behavior's other operations, as well as port assignment for the array operations.

There are interactions between the memory design and the scheduling and allocation of other operations, which are ignored by our segmented design flow. For instance, if the results of two different operations are stored into two different arrays and those operations share the same functional unit, then grouping the arrays in the same memory would allow both operations to share the connection between the functional unit and the memory.

4.2 Compound moves

The basic moves are sufficient to reach every design possible with our representation, but are not a good move set for an annealing algorithm. Using the basic moves, the path between two legal designs (designs that don't violate any constraints or penalties) may include many illegal designs. Illegal designs are rarely acceptable in the later stages of annealing and therefore a full exploration of the design space is difficult. Our move set has been improved with the addition of the following compound moves.

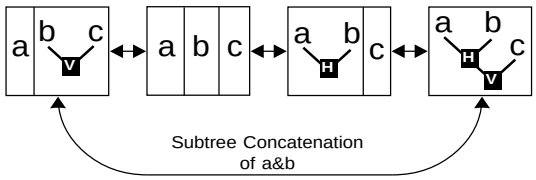

Figure 5. Subtree grouping

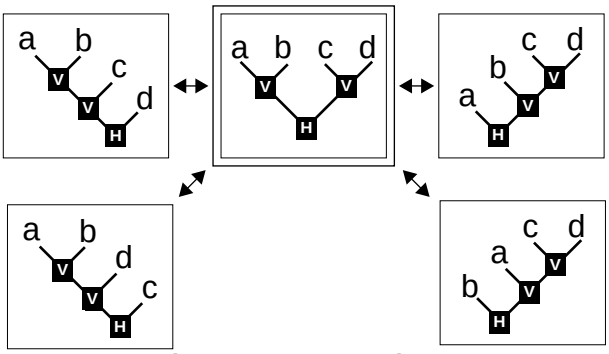

Figure 6. Tree rotations

Subtree grouping and ungrouping: The basic array grouping operations only modify the top of the array group tree so in order to modify any subtree it is necessary to disassemble much of the tree using the ungrouping move and then reconstruct the tree in the desired way. An example of this procedure is shown in Figure 5, where we want to horizontally concatenate the **a** and **b** arrays together, while retaining the vertical concatenation to array **c**. We have extended the grouping and ungrouping operations to apply to subtrees so that modifications like that shown below can be accomplished in one move.

Grouping operation modification: With the basic move set, changing the type of array grouping operation at a particular node requires disassembling the tree and reconstructing it with the desired grouping operation. To avoid this we allow direct modification of the grouping operation at any node in the tree.

Tree rotation: To modify the internal tree structure without disassembly, we use AVL binary tree rotations. In Figure 6 we show a tree and the four possible AVL rotations.

Heuristic scheduling: Modifying the schedule is difficult using the basic move set, because the path between two legal schedules may include many illegal schedules. To overcome this problem, we have implemented a set of scheduling heuristics that can be employed by the annealer to schedule all accesses in a schedule set. The particular heuristic used for each schedule set can be modified by the annealer. This approach is supported by [3], which showed that annealing with a basic move set always generates optimal or near-optimal schedules, but the result from a large number of scheduling heuristics will usually find the optimal quicker.

5.0 Examples

To demonstrate the effectiveness of our array mapping approach, we have synthesized two memory intensive applications: a speech phoneme recognizer based on Viterbi search [14] and a fuzzy controller [12]. In both examples, the memory component library contains a small on-chip memory (64 words by 8 bits) and three large memory components: 8K by 8, 32K by 8 and a 128K by 8. The number of the small memories is limited by the amount of on-chip area that can be budgeted for memory for a particular application. The number of larger memory components is unlimited, but we have assumed that the cost of a 128K x 8 SRAM is two times that of a 32K x 8 SRAM, which is two times the cost of a 8K x 8 SRAM.

The Viterbi search application consists of only one basic block that accesses arrays, so memory performance is solely determined by the schedule length of that basic block. This behavior contains seventeen small arrays and eleven 16K x 8 bit arrays, which are all read once with the same index in each iteration of the basic block. The small arrays are consistently mapped to the small on-chip RAMs. We will discuss the mapping of the eleven large arrays.

The design space exploration of this application was conducted by varying the relative weight of memory and performance in the system cost equation and by using some performance constraints. Memory wires were weighted very lightly in the cost equation so that they are reduced as long as there is no impact on memory cost or performance. The results of this exploration are illustrated in Figure 7 and their cost is plotted versus schedule length in Figure 8.

In design A, the eleven arrays are all in one array group, which has been mapped to six 32K x 8 RAMs. Design A has a two cycle schedule, which is the fastest possible for this application using the given data path technology. Two particular arrays have accesses that are on the critical path in the behavior and must both be accessed in the first cycle. To allow this, the design optimizer has placed both these arrays in the same rank so they may both be read together in the first cycle. This illustrates the coupling between the data dependencies in the behavior and the desired memory design of the implementation.

In the remaining designs, memory cost is reduced by using more of the 128K RAMs, which have a lower cost per bit. Designs B, D, and F are in some sense inferior because they sacrifice performance without reducing memory cost. However, these designs have fewer memory wires than the design with equivalent memory cost and higher performance. This wire reduction was obtained either by favoring a more vertical arrangement of arrays to reduce data bus size, as in design B and F, or by using horizontal concatenation to reduce address bus size, as in design D. Design E and F have the minimum memory cost. Therefore our tool is capable of generating implementations for this application that span the entire memory design space.

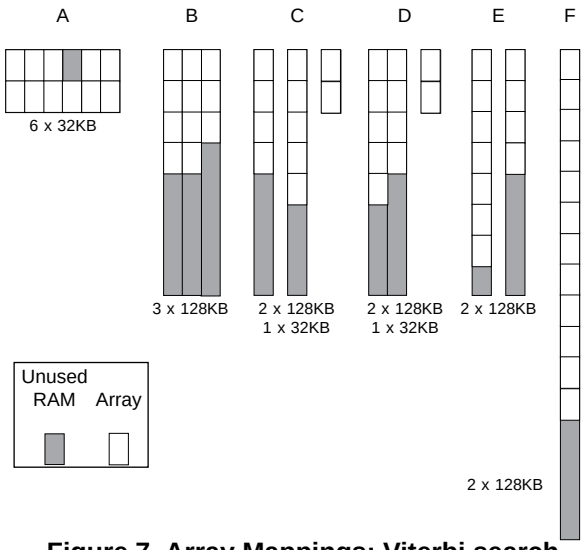

Figure 7. Array Mappings: Viterbi search

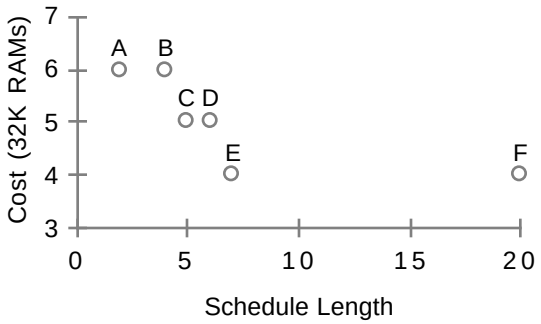

Figure 8. Cost vs. schedule length: Viterbi search

We have conducted a similar design exploration of a fuzzy controller application. This application has three basic blocks that access memory, one of which initializes a long computational loop and is rarely executed. Of the two remaining basic blocks, one is executed approximately twice as often as the other, so we have created a expression of performance based on the weighted schedule length of these blocks. There are twenty arrays with various dimensions, which are all too large to fit onto on-chip memory. The results of this exploration are illustrated in Figure 9 and their cost is plotted versus the weighted schedule length in Figure 10.

As performance is reduced, memory cost is again reduced by using larger RAM components, which have a lower cost per bit. The number of memory wires is reduced by having fewer address busses and narrower data

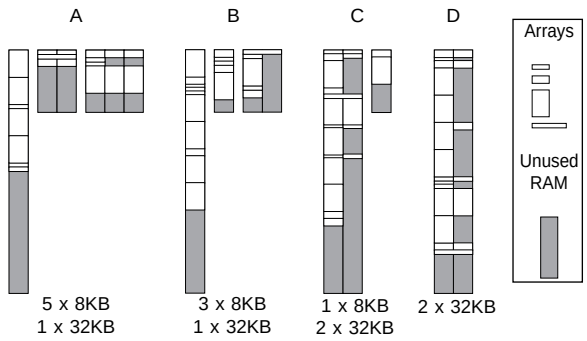

Figure 9. Array mappings: fuzzy controller

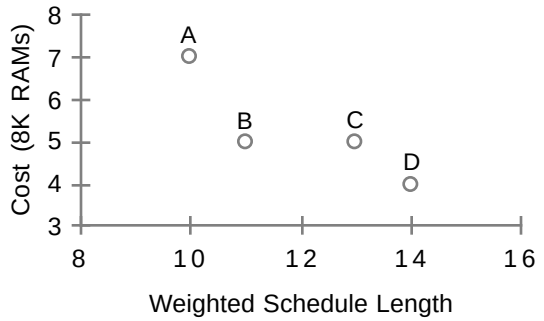

Figure 10. Cost vs. weighted schedule length: fuzzy controller

busses. When horizontal concatenation is used, similarly accessed arrays are lined up so they can be simultaneously accessed. Design A has the highest performance possible with the given data path technology and memory component library. This was verified by mapping every array to a separate 8K RAM in order to obtain the fastest access time and eliminate all resource conflicts. The best possible schedule lengths for that binding equaled those of design A. Design D has the lowest memory cost possible for this application.

6.0 Summary

We have described our approach to the problem of mapping the arrays in a behavior to memories in an implementation of that behavior. Our design representation is based on grouping arrays together and binding the array groups to memory components. Because the problem of determining an appropriate array-memory mapping is coupled with the problem of scheduling the array access, our design tool integrates access scheduling with our array grouping and memory binding techniques. We have used this design tool to synthesize two memory-intensive applications, and have demonstrated its ability to create designs that span the entire memory design space: from the highest performance design possible with the given data path technology to the lowest memory cost design possible with the given memory component library.

7.0 Acknowledgments

The authors thank the Semiconductor Research Corporation (under contract DC-94-068) and the National Science Foundation (under contract MIP-9112930) for funding this research.

8.0 References

[1] M. Balakrishnan, A. Majmudar, D. Banerji, J. Linders, and J. Majithia, “Allocation of Multiport Memories in Data Path Synthesis,” *IEEE Trans. on CAD*, April 1988.

[2] R. Cloutier and D. Thomas, “The Combination of Scheduling, Allocation and Mapping in a Single Algorithm,” *Proc. of the 27th DAC*, pp. 71-76, 1990.

[3] C. Coroyer and Z. Liu, *Effectiveness of Heuristics and Simulated Annealing for the Scheduling of Concurrent Tasks: an Empirical Comparison*. INRIA report 1379, 1991.

[4] D. Karchmer, and J. Rose, “Definition and Solution of the Memory Packing Problem,” *Proc. of ICCAD*, pp.53-58, San Jose, CA, Nov. 1994.

[5] T. Kim and C.L. Liu, “Utilization of Multiport Memories in Data Path Synthesis,” *Proc. of DAC*, 1993.

[6] P. J. M. van Laarhoven and E. H. L. Aarts, *Simulated Annealing: theory and applications*, Dordrecht, Boston, 1987.

[7] D. Maydan, S. Amarasinghe, and M. Lam, “Array Data-flow Analysis and its Use in Array Privatization,” *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 2-15, 1993.

[8] E. Ochotta and T. Mukherjee, “Programmer’s Guide to the Reconfigurable Simulated Annealing Library (anneal),” Report No. CMUCAD-94-35, Carnegie Mellon University, August 1994.

[9] T. Peng, and D. Padua, “Automatic Array Privatization,” *6th International Workshop on Languages and Compilers for Parallel Computing*, pp. 500-522, Springer-Verlag, Berlin, 1994.

[10] L. Ramachandran, D. D. Gajski, and V. Chaiyakul, “An Algorithm for Array Variable Clustering,” *Proc. of European Design and Test Conference (EDAC)*, 1994.

[11] H. Schmit and D. E. Thomas, “Array Mapping for Behavioral Synthesis,” Report No. CMUCAD-94-46, Carnegie Mellon University, October 1994.

[12] H. Schmit and D. E. Thomas, “Implementing Hidden Markov Models and Fuzzy Controllers on FPGAs,” *IEEE Symposium on FPGAs for Custom Computing Machines*, April, 1995.

[13] D. E. Thomas, et al. *Algorithmic and Register-Transfer Level Synthesis: the System Architect’s Workbench*, Kluwer Academic, 1990.

[14] D. E. Thomas, J. Adams, and H. Schmit, “A Model and Methodology for Hardware-Software Codesign,” *IEEE Design and Test of Computers*, Vol. 10, No. 3, pp. 6-15, Sept. 1993.

[15] J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goossens, and H. De Man, *High-Level Synthesis for Real-Time Digital Signal Processing*, Kluwer Academic Publishers, Norwell, MA, 1993.

[16] D. F. Wong and C. L. Liu, “A New Algorithm for Floorplan Design,” *Proc. of the DAC*, pp. 101-106, 1986.