A Simple Placement and Routing Algorithm for a Two-Dimensional Computational Origami Architecture

Robert S. French

April 5, 1989

Abstract

Computational origami is a parallel-processing concept in which a regular array of processors can be folded along any dimension so that it can be simulated by a smaller number of processors. The problem of assigning functions to each of the processors is very much like the generalized electrical circuit layout problem. This paper presents a simple, polynomial time algorithm for placing and routing functions in an origami architecture. Empirical results are analyzed and optimizations suggested.

1 Introduction

Computational origami is a parallel-processing concept developed by Alan Huang [7, 8, 9]. An origami machine consists of a regular array of processors superimposed on a tessellable mosaic architecture [11]. The array can be arbitrarily ‘‘folded’’ along any of its dimensions so that it can be simulated by a smaller number of processors. The latency and throughput of the system can be adjusted by folding the array widthwise or depthwise, respectively. The folding of an origami array is transparent to the software running on it. Information about the construction of an origami machine can be found in [3].

A sample origami array is shown in figure 1. This is a two-dimensional array of processors with each processor having two inputs and two outputs. The outputs of a processor are staggered with the inputs of the processors on the next row so that signals can be distributed as necessary. Each processor contains no state and performs one simple operation per cycle. Each can take on one of a number of *flavors*, or operations that can be performed. All processors have the same selection of flavors. Thus, an origami array is programmed simply by selecting a flavor for each of the processors. The processors in an origami array are called *nodes*. Note that data always flows ‘‘down’’ an origami array, and never up or directly sideways.

While there are many different interconnection strategies that can be used, and the origami concept can be easily extended to more than two dimensions, we will only treat this simplified model here. Information about other architectures can be found in [11].

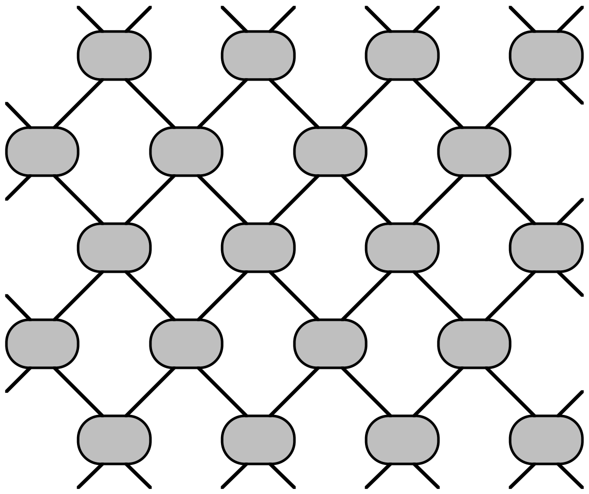

Figure 1: A sample two-dimensional origami array.

It is easy to see that any logical function can be computed with an appropriate array of nodes. For example, the array illustrated in figure 2 takes six inputs, performs a logical AND on them, and produces a single output. The array illustrated in figure 3 implements a 4-bit by 4-bit adder using half-adders (HA), OR gates, and routing elements (an HA element produces the sum on its left output and the carry on its right output).

The primary problem with an origami system is assigning functions to each of the nodes so that a task can be per-

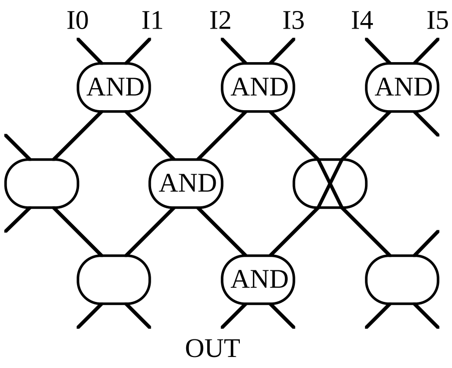

Figure 2: A simple 6-input AND tree.

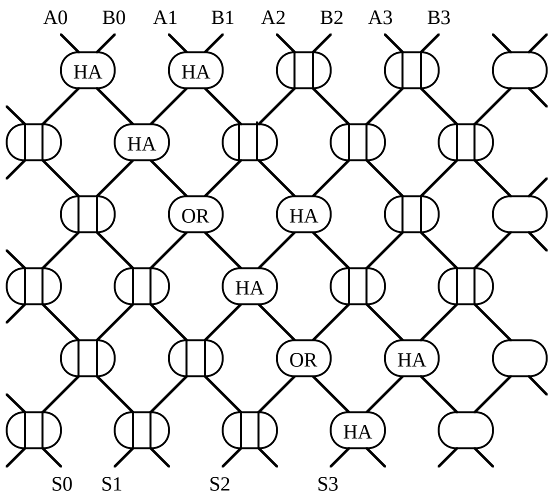

Figure 3: A 4-bit by 4-bit adder made from half-adders and OR gates.

formed efficiently with a minimum amount of hardware. Standard compiler techniques can be utilized to generate dataflow graphs from computer languages, but creating an origami array from the logic functions produced is a difficult problem. In many ways this is analogous to the generalized two-dimensional electrical circuit layout problem (*e.g.* the automatic routing of wires on a printed circuit board), but it is sufficiently different that circuit layout algorithms are not directly applicable. Finding an optimal placement and routing is considered to be NP-complete, although this has not been proven formally.

Chuang [2] developed a prototype routing system for a three-dimensional origami array using a flooding algorithm with backtracking. This was used to place and route a Wallace tree adder. This paper proposes an efficient algorithm for placing logic functions and routing between them without the need for backtracking. Because the result can be optimized in an iterative fashion, the algorithm can be run for a predefined time and then the best result achieved so far can be returned.

2 Preliminaries

For the purposes of this algorithm, an origami array is a two-dimensional staggered array of nodes as described above. In a fully automated logical compiler it might be desirable to treat each node independently for placement. However, in a large application the number of nodes may be very large (equal to the number of discrete logical functions that need to be performed) and compilation time quickly increases beyond the realm of practicality. Therefore it is frequently desirable to break down the problem into subproblems, and place and route the nodes required for each subproblem separately or perhaps even by hand. Once the placement and routing for a subproblem has been produced, we can consider the result to be an atomic unit for future connection to other nodes and subproblems. Such a subproblem is called a *module*, and is considered in this algorithm to be an by rectangular set of nodes with defined input and output locations. Typical modules are -bit adders, multipliers, and bus multiplexors. Individual ungrouped nodes are also considered to be modules.

This algorithm assumes that there are four flavors dedicated to routing. They are:

1. *passthrough*: copy the left input to the left output, and the right input to the right output.
2. *crossover*: copy the left input to the right output, and the right input to the left output.
3. *left broadcast*: copy the left input to both the left and right outputs, and discard the right input.
4. *right broadcast*: copy the right input to both the left and right outputs, and discard the left input.

Procedures for breaking a problem into subproblems will not be discussed in this paper.

Their symbols are listed in figure 4. A series of connected routing flavors is called a *wire*.

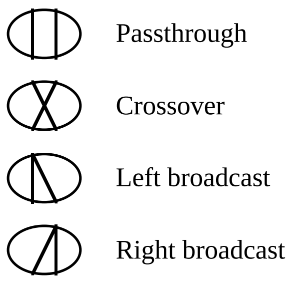

Figure 4: Flavors used for routing.

The algorithm requires the following data as input:

- , the set of all of the modules which need to be placed.
- , the set of dependencies between the modules. Each corresponds to and is the set of modules which provide inputs to .
- , the set of routings between the modules. Each indicates an output pin of a module (*src*) and the input pin of another module (*dest*) it should be connected to.

Inputs and outputs to the array are treated as special modules which are used during placement and routing, but are not actually placed in the physical array.

3 Module Placement

The first step in the algorithm is to place the modules in an array so that routing between them is as short as reasonably possible. Modules are first ordered vertically into distinct *levels* (this is a *logical* placement—the physical placement won't be determined until later), and then are placed horizontally within each level.

The ordering of modules vertically is a simple process:

1. Mark all of the modules as unused.
2. Add all of the inputs to level 0, and mark them as used; set the level number to 1.
3. Add to the current level all modules (which are not outputs) which depend only on modules which have already been added to previous levels. That is, all modules such that isn't marked as used, and for each module , is marked as used and the level of is less than the current level.
4. Increment the level number by 1.
5. If all modules which are not outputs are marked as used, add the outputs to the current level and stop.
6. Go to step 3.

Note that step 3 must eventually use all of the modules because there can be no circular dependencies between modules (the modules form a strict hierarchy).

Once the modules have been ordered vertically, they must be arranged horizontally. This is done in two stages: ideal placement, and shifting to allow room for routing. During the ideal placement stage, the modules are placed in such a way that the total idealized routing distance to each module is minimized. For a given module , the idealized routing distance is the sum of the squares of the horizontal displacements for each module that feeds data to an input of . The horizontal position of these modules will always be known since module placement proceeds in order down the hierarchy. For each level, modules are picked one by one and placed in such a manner that their idealized routing distance is minimized and they do not overlap. This can be done very efficiently using a variant of the median method discussed in [5].

Once the modules are ordered vertically and placed horizontally in an ideal manner, some may need to be shifted to make room for wires which need to go between modules. This is done by tracing the ideal path of each wire while keeping counters (which we will call *right* and *left*) indicating how much space adjacent modules need between them and creating a *goal* for each wire on a per level basis. The algorithm is:

1. For all , let *right* *left* .
2. For each , follow the routing from its source to its destination in a straight diagonal line. Whenever would intersect a module, , increment either *right* or *left* depending on whether the wire is intersecting the right half or left half of the module, respectively. Keep track of which modules needs to pass between, and whether it needs to pass on the right or the left.

3. Shift the modules (keeping their same relative horizontal position) such that the distance between adjacent modules and is at least *right* *left*.

4. For each , find the new position of each pair of modules it is going to pass between, and assign it a goal column for that level such that no two wires have the same goal column for that level (there must be enough space because of steps 2 and 3). The goal column for the wire's starting level is the horizontal position of the appropriate output of the source module, and the goal column for the destination level is the horizontal position of the appropriate input of the destination module.

Once this step is completed, all modules have been placed in such a manner that routing, using the appropriate goal columns, *must* be possible without backtracking.

4 Routing

Now that the modules have been placed horizontally and ordered vertically, and each wire that needs to be routed has a goal column for every level it must pass through, routing can proceed. Routing proceeds from the outputs, up through the levels, to the inputs (thus routing proceeds in the *opposite* direction from the flow of data). The routing algorithm maintains the following state: the current level and the set of wires, , which are currently being routed. It proceeds as follows:

1. Set equal to all whose destination module is an output, and set each wire's current position to the horizontal position of the output. Set the current level to the level on which all outputs reside minus 1.

2. Determine the goal column for each for the current level.

3. Route each wire toward its goal column by the method outlined below, and continue until all wires have reached their goal columns.

4. Place all modules which reside on the current level at their desired horizontal position, and delete all whose source module has now been placed.

5. Continue routing as in step 3. Whenever the top of a freshly placed module is reached, add the wires whose destinations are that module to . Continue until the tops of all modules on this level have been reached.

6. Decrement the current level number.

7. Repeat until level 0 (the level containing only array inputs) is reached.

8. Continue routing wires until all wires have reached their goal columns (the positions of the appropriate inputs).

As wires are being routed, a number of conflicts can arise. These include two wires interacting (such as needing to cross) or a wire needing to move left or right and not being able to (because of interconnection constraints). Such conflicts are resolved according to the appropriate entry in table 1. For example, if the wire entering on the left side of the node needs to go right and the wire entering on the right side of the node needs to go left, a crossover should be placed at the current location. Likewise if only one wire is entering the node, is entering on the right, and needs to go left, a crossover should be placed.

The only conflict not covered by this table is the case where two wires are entering a node and have the same goal column for the current level. In this case, the wires should be combined by using a left or right broadcast and removing one of the wires from the current wire list .

5 Performance Analysis

All portions of the presented algorithm run in polynomial time in the number of modules. The estimates given below are easily achieved with standard programming practices, and better upper bounds can probably be achieved with a little effort. Specifically, assuming there are modules:

Ordering the modules vertically is an operation.

Ideal horizontal placement of modules is .

There are wires, and thus the process of shifting the modules horizontally is .

There are wires, and the array is nodes high, so routing is in general (although it is actually slightly worse).

		Wire on left needs to go			
		left	straight	right	none
Wire on right needs to go	left	passthrough	crossover	passthrough	crossover
	straight	passthrough	passthrough	crossover	passthrough
	right	passthrough	passthrough	passthrough	passthrough
	none	passthrough	passthrough	crossover	

Table 1: Flavors used to resolve various routing conflicts.

The algorithm was implemented and included in a simple compiler [4]. Table 2 shows the running times on a DEC VS2000 workstation for this algorithm for the generation of ripple-carry adders with 8–20 bits of input (4–10 bits for each operand) and selectors for 2–20 bits (generating 1–10 selected bits with a single select line). As can be seen, for specific applications the algorithm can run in almost linear time.

Function	# of modules	Time (sec)
Adder (8 bit)	5	2.1
Adder (12 bit)	7	4.4
Adder (16 bit)	9	6.4
Adder (20 bit)	11	8.5
Selector (2 bit)	4	.9
Selector (6 bit)	12	1.7
Selector (10 bit)	20	3.3
Selector (20 bit)	40	10.1

Table 2: Placement and routing times for several sample applications.

6 Optimizations

Unfortunately, we pay for the speed of the algorithm with inefficiencies in the resulting origami array. In the applications that have been generated by this algorithm so far (including a few simple 8-bit 3-function calculators and a 16-point convolution machine), routing accounts for approximately 45% of all assigned nodes, while another 45% of the nodes are left unused entirely. When we consider that in an ideal situation each node would have a physical piece of hardware associated with it, and that the latency of the system is proportional to the height of the array, we can see that this is a tremendous amount of wasted time and hardware. Most of the specific pathological cases that have been encountered are too complex to be discussed in this paper, but there appear to be a number of general ways to improve this algorithm.

One routing problem arises when two buses need two cross. For single wires this is obviously not a problem, but for large buses of wires to cross huge areas may have to be dedicated to routing. This amount is greatly increased when the wires in the bus are densely packed (they have no space on either side). Adding the constraint that, when not required to be densely packed to interface to a module, wires should have at least one free space between them should significantly decrease the amount of routing required.

Many other optimizations can be applied in an iterative manner. A popular way to do this is called *simulated annealing* [10, 1, 6, 12]. In this method, one of a number of optimizations is chosen and applied to the current system. A cost function is used which indicates the desirability of a given system, and the change in cost ( ) from the original system to the new system is computed. The new system is accepted with probability:

where T is the ‘‘temperature’’ of the system which gradually decreases as more optimizations are applied, thus accomplishing an ‘‘annealing’’ effect. Two of the optimizations which could be iteratively applied using this or similar methods are:

The modules have been placed to minimize the idealized routing distance; however, it is possible that they have not been placed to minimize the real routing distance. Localized transposition of modules should decrease routing requirements in many cases.

An experimental implementation of this optimization achieved an over 40% reduction in array size.

This can be done using a method similar to the –neighborhoods presented in [5].

Much of the routing in an origami array is devoted to permuting a set of wires to match the input requirements of a module. For example, if an adder has an output with the most significant bit on the left, and this result needs to be sent to a negation module which expects the most significant bit on the right, a great deal of time will be spent rearranging the wires to satisfy this constraint. While it is possible to partially solve this problem by developing libraries of ‘‘matching’’ modules, it is impossible to do this for all combinations in practice. A simple solution to this problem is to provide more than one module capable of performing a particular task. The modules would be identical in function, but would have their input and output pins permuted in different ways so that the routing distance could be reduced by selection of the appropriate modules. Since it is impossible to determine which instance of a module will produce the largest reduction in array size, modules must be chosen at random during the annealing process.

None of these optimizations have been fully implemented at the time of this writing.

7 Conclusion

An algorithm to place and route logic modules in an origami array has been developed. The algorithm runs in polynomial time in the number of modules, and can achieve almost linear performance in some cases. However, the resultant origami array is very inefficient and consists primarily of routing and unassigned nodes. Several iterative optimization techniques were presented including techniques based on simulated annealing, but none have been implemented at the time of this writing.

References

[1] ČERNÝ, V. A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications 45*, 1 (Jan. 1985), 41–51.

[2] CHUANG, I. L. Computational origami. Aug. 1988.

[3] CHUANG, I. L. An introduction to the application of computational origami. Feb. 1989.

[4] CHUANG, I. L., AND FRENCH, R. S. Karma I: An origami architecture computer. Dec. 1988.

[5] GOTO, S. An efficient algorithm for the two-dimensional placement problem in electrical circuit layout. *IEEE Trans. Circuits Syst. CAS-28* (Jan. 1981), 12–18.

[6] GROVER, L. K. A new simulated annealing algorithm for standard cell placement. In *Proceedings IEEE International Conference on Computer-Aided Design* (1986), pp. 378–380.

[7] HUANG, A. Architectural considerations involved in the design of an optical digital computer. *Proceedings of the IEEE 72*, 7 (July 1984), 780–786.

[8] HUANG, A. Computational origami. Patent application, July 1987.

[9] HUANG, A. Computational origami - the folding of circuits and systems. In *Proceedings of the 1989 Optical Computing Conference* (Feb. 1989). To appear.

[10] KIRKPATRICK, S., GELATT, JR., C. D., AND VECCHI, M. P. Optimization by simulated annealing. *Science 220*, 4598 (May 1983), 671–680.

[11] LU, H. Computational origami: A geometric approach to regular multiprocessing. Master’s thesis, MIT Department of Electrical Engineering and Computer Science, May 1988.

[12] WONG, D. F., LEONG, H. W., AND LIU, C. L. *Simulated Annealing for VLSI design*. Kluwer Academic Publishers, 1988.