Using Simulated Annealing for Mapping Algorithms onto Data Driven Arrays [1]

Bilha Mendelson
IBM Israel
Science & Technology
Haifa 32000, Israel

Israel Koren
Dept. of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003

Abstract Control-driven arrays provide high levels of parallelism and pipelining for inherently regular computations. Data-driven arrays can provide the same for algorithms with no internal regularity. The purpose of this paper is to establish a method for mapping any given algorithm onto data driven array. The method, based on the simulated annealing algorithm, aims to find an efficient mapping that minimizes the area, and maximizes the performance of the given algorithm or find a tradeoff between the two.

1 Introduction

A variety of topologies and architectural designs of processor arrays have been proposed and many computational algorithms for these arrays have been devised [4]. These include globally synchronous systolic arrays and globally asynchronous wavefront arrays [3],[6]. Most existing algorithms for these arrays were developed for problems with inherent regularity. These computations proceed in the processor array in a predetermined manner and achieve high performance through parallelism and pipelining.

There are however, many computationally demanding problems which do not exhibit high regularity. The data-flow mode of computation seems to be an appropriate approach to follow. We adopt therefore, the approach proposed in [5] where the algorithm is first represented in the form of a data flow graph (DFG) and then mapped onto a data driven processor array. The processors in this array will execute the operations included in the corresponding nodes (or subsets of nodes) of the DFG, while regular interconnections of these elements will serve as edges of the graph. The data driven processor array is a programmable and homogeneous array which is composed of an hexagonal mesh of identical processing elements (PEs). The hexagonal topology of the array serves only as an example of a regular structure.

A data driven PE is capable of performing arithmetic, logical, routing and synchronization operations, and can contain more than one operation. In particular, the routing operations can be performed in parallel to the arithmetic and logic operations.

A single connection link between adjacent PEs in the array was shown to be a critical limitation [11, 1]. It affects both the communication between nodes in different PEs and the capability to include more operations in the same PE. We follow an alternative approach: two communication links connect every two adjacent PEs. Since each input and output operand is held in an internal register, the number of nodes assigned to a PE is limited by the total number of registers in the PE. In our design each PE contains 16 internal registers (12 for communicating with its six neighbors and 4 scratchpad registers).

Figure 1 shows a data–flow graph representing the computation of the two coefficients A and B in the solution $Y(t) = A\cos wt + B\sin wt$ of a spring-mass system with an external force $F(t) = F_0\cos wt$. Figure 2 depicts a possible mapping of the graph in Figure 1 onto a regular processor array.

A method for mapping a given algorithm onto a data driven array to achieve a better performance has been developed. The mapping involves assigning every node of the DFG to a data driven PE in a data driven processor array. The goal of an efficient mapping is to minimize the

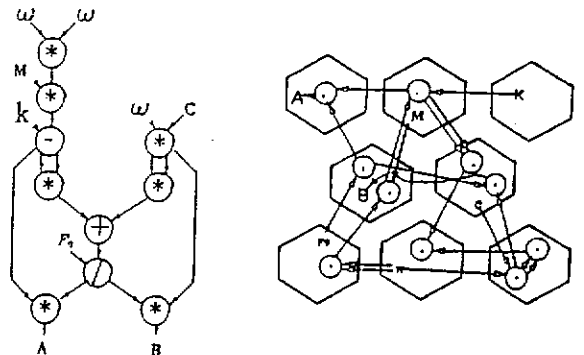

Figure 1: A data flow graph.

Figure 2: The mapping of the graph in Figure 1 onto a hexagonally connected array.

area (i.e., the number of PEs used) as well as optimize the performance (pipeline period and latency) of the given algorithm, or find a tradeoff between the area, pipeline period and latency criteria. The search for an optimal mapping is a combinatorial optimization problem.

Simulated annealing is known to be a powerful algorithm for solving combinatorial optimization problems [7]. Section 2 describes the use of the simulated annealing for mapping DFGs. The simulated annealing procedure may become very time consuming when applied to large problems. Section 3 shows how to deal with large programs by dividing the given algorithm into several blocks and mapping each of them separately. Later, these blocks will be merged together in order to put them as close to each other as possible. Conclusions and future research are presented in the last section.

[1] This work was done while B. Mendelson was with the University of Massachusetts at Amherst and was supported in part by SRC under contract 90-DJ-125.

2 Simulated Annealing and the Mapping Problem

Simulated annealing (SA) was shown to be a powerful algorithm for solving combinatorial optimization problems [7]. The SA algorithm is used in VLSI design [12] (for placement, floorplan design, channel routing, etc.) and in the design of multicomputer systems. Johnson et al. [2] have compared the performance of the SA algorithm and other heuristic algorithms for the graph bisection problem. Their conclusion is that if the given problem has a regular structure, it is likely that a good heuristic algorithm will yield good results. However, when there is no such structure it is better to use SA.

Finding a mapping for an algorithm that minimizes area and optimizes performance is an NP-complete problem. The DFG for an arbitrary algorithm has no particular structure and it is very difficult to find a good heuristic algorithm for generating an optimal mapping. Therefore, we choose to use the SA algorithm. This method aims to find an efficient mapping that minimizes the area, and maximizes the performance of the given algorithm or find a tradeoff between the two.

To use the SA algorithm for the mapping process we have to define some basic parameters that the process needs. These include the initial configuration [2], acceptable moves and a cost function. The initial configuration is some feasible mapping where every DFG node is assigned to a PE (the simple mapping procedure presented in [5] can be used to generate such an initial configuration).

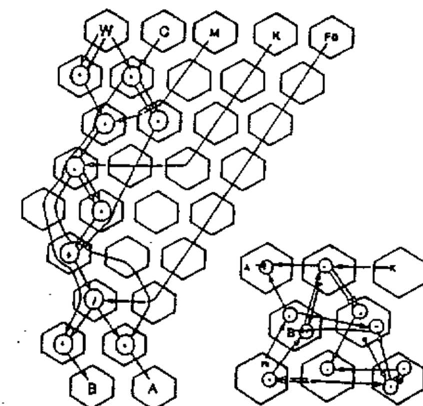

Figure 3: Mapping using the simulated annealing process.

Acceptable moves that change a configuration are: move a node from one PE to another and exchange nodes between PEs. The first move allows the nodes to move from one PE to another PE which has not used all its capacity, while the purpose of the second move is to allow reassignment of nodes even for PEs that used all of their capacity.

A result of such a mapping is shown in Figure 3. Figure 3(a) shows the initial configuration and Figure 3(b) the result after performing the SA process. In this example the goal was to minimize the area. Notice that the area has decreased by 60%, which is a substantial improvement over the initial configuration.

[2] A certain mapping of the DFG onto the hexagonal array is called a *configuration*.

In most applications, there is a need to combine area minimization and performance maximization. We suggest therefore, three criteria for evaluating a mapping:

Area - number of PEs utilized by the mapped DFG.

Pipeline period - mean time between successive results.

Latency - time elapsed from entering the input operands until the output is produced.

A good mapping is one that optimizes all three criteria according to the weight coefficients that are supplied to the SA algorithm. The cost function, which is used to evaluate the new configuration is given by $\lambda_a A + \lambda_p P + \lambda_l L$ where A is the area (number of PEs) being used, P is the estimated pipeline period, L is the estimated latency and λ_i's are weight coefficients ($\Sigma\lambda_i = 1$, $i = a, p, l$).

Minimizing the area is achieved by compressing several DFG nodes into the same PE. The nodes that are ready to be executed and are assigned to the same PE are executed sequentially. The pipeline period of the mapped algorithm is therefore,

$$P = \max_E\left(\sum_{i=1}^{n_i} EX(node_i)\right)$$

where $EX(node_i)$ is the execution time of node i, n_i is the number of nodes assigned to PE_i and E is the set of all PEs that some node has been assigned to them. The pipeline period of the mapped algorithm may be affected by the way the compression is done. It can increase if in the assignment process the total execution time of nodes assigned to the same PE exceeds the estimated pipeline period. The chosen weight coefficients determine the kind of optimization that is performed.

The temperature in the SA procedure controls the probability of accepting a new configuration, which not necessarily reduces the cost function. This temperature is reduced geometrically by the factor r, called cooling rate. Figure 4 shows the average number of iterations needed by the SA algorithm to obtain the final mapping for four different values of $(\lambda_a, \lambda_p, \lambda_l)$. As the cooling rate increases, the temperature reduces slower. Therefore, as was expected, in all the cases the number of iterations increases as the cooling rate increases. We can observe from the figure that the number of iterations is sensitive to the value of λ_l. The average number of iterations increased by 71.1% when λ_l was changed from 0 to 0.3 (because additional routing nodes are needed when a node is moved from one PE to another). Adding routing nodes affects the latency more than the pipeline period and the area, and therefore, the algorithm needs more iterations to converge.

Using a performance estimation method for evaluating the potential performance of algorithms for data driven machines [9] (and the execution times from [1]), we know that for the example in Figure 1 the optimal pipeline period and latency are 16 and 50, respectively. By applying an exhaustive search for the best mapping, we found out that the minimum number of PEs needed for the optimal mapping is 7. We denote the probability to get the best achievable mapping by p. SA is a random algorithm and therefore, there is a need to perform several runs and choose the best mapping. The probability to achieve a good mapping in n runs is:

$$P_n = 1 - (1 - p)^n$$

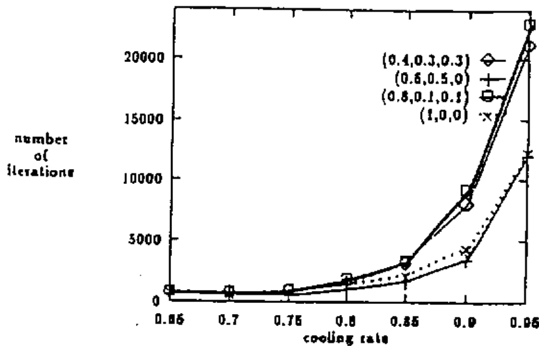

Figure 4: Average number of iterations needed by the SA to obtain a mapping of the algorithm in Figure 1.

From this expression we obtain the minimum number of runs of the SA algorithm to ensure a good mapping with probability P_n.

Table 1 shows the minimum number of runs needed to get a good mapping for $P_n = 0.95$ and different values of $(\lambda_a, \lambda_p, \lambda_l)$. In order to obtain the entries in this table, we have run the SA on the example many times. The mappings with area and performance no worse than the desired area and performance were counted. (A value of ∞ indicates that a good mapping was never found.) The probability to get a good mapping in a single run, p, was calculated by dividing the total number of good mappings by the total number of simulated annealing runs.

We used the execution times from [1] (i.e., divide takes 16 clock cycles, multiply 8 clock cycles and the rest take one clock cycle). Column 3 of the table gives the number of runs that are needed in order to achieve the global minimum mentioned above which uses only 7 PEs and pipeline period and latency of 16 and 50, respectively. Column 4 (5) shows this number for a mapping which uses at most 8 (9) PEs, pipeline period of 17 (19) and latency of 55 (60).

It can be seen that there is a need for few runs (2-5) of the SA algorithm, in order to achieve the optimum area and performance, if we select a cooling rate of 0.95. If the use of 8 PEs and reduction of 10% in the performance is allowed, only one run in most of the cases is needed.

When only one run of the SA algorithm is allowed, there is a need to know the average performance of this method. Figure 5 shows the mean value of the area, pipeline period and latency obtained for the algorithm in Figure 1.

Table 1: Number of runs needed to get a mapping with $P_n = 0.95$ for the algorithm in Figure 1.

$(\lambda_a, \lambda_p, \lambda_l)$	Cooling rate	15 PEs pip. period=8 latency=15	18 PEs pip. period=10 latency=18
(0.4,0.3,0.3)	0.90	∞	∞
	0.95	∞	273152.5
(0.5,0.5,0)	0.90	∞	∞
	0.95	∞	130709.8
(0.8,0.1,0.1)	0.90	∞	∞
	0.95	∞	373269.6
(1,0,0)	0.90	∞	∞
	0.95	331527.2	53956.0

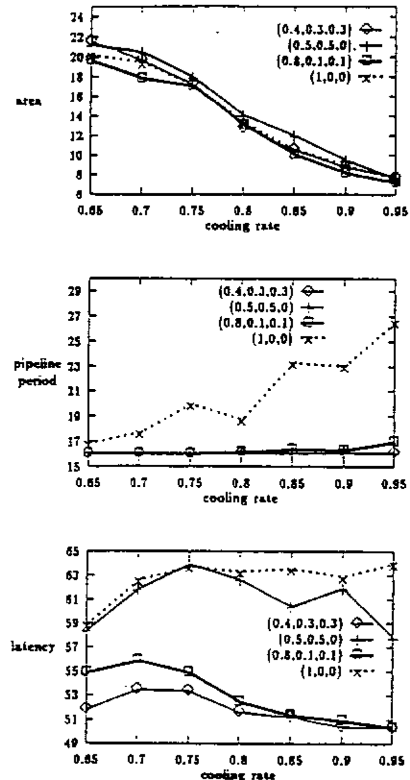

Figure 5: Average area and performance achieved when mapping the example in Figure 1.

We have mapped several algorithms onto data driven arrays and although our experience with the SA algorithm is limited, we can still recommend ways to choose the parameters of the SA. The best performance and area minimization is achieved when $r = 0.95$. It is true when considering the average behavior of SA as well as when calculating the number of iterations needed to get the best achievable mapping. In some cases, when $r < 0.95$, the SA uses a larger number of iterations without improving the mapping. Therefore, a cooling rate of 0.95 is advised for mapping graphs.

As we increase λ_a and λ_p, the effect of the change on the pipeline period is not significant. However, it has a significant impact on the area. Since in our examples the pipeline period is almost insensitive to the value of λ_p, area minimization can still be achieved without adversely affecting the pipeline period by slightly reducing the pipeline period coefficient and slightly increasing the area coefficient.

In summary, SA proved to be an efficient procedure for mapping algorithms onto homogeneous arrays. If $r = 0.95$ is chosen, only a few runs of the SA algorithm are needed to obtain a satisfactory mapping. It was also proved that area minimization can be obtained in addition to performance maximization by choosing the right weight coefficients.

2.1 Improvements of the Initial Mapping

In the previous section, we presented the result of the SA algorithm adapted to the mapping problem. These results have been obtained by applying the SA algorithm to an initial mapping that was done using the method presented in [5] (e.g., assigning a single node to a PE). We have found that if we try to put several nodes together in the same PE after the initial mapping has been done, the quality of the SA results improves.

This compaction phase is done by shifting the DFG nodes in six directions (down-right, down-left, left, up-left, up-right and right), as will be explained in the next section. The idea is to minimize the number of unutilized PEs in the mapped array.

3 Mapping large algorithms

When the number of nodes of the DFG is sufficiently small the SA algorithm gives satisfactory results. However, when the number of nodes increases, an optimal solution is not reached. But still the results of the mapping process are good and close to the optimum. Besides the degradation in the quality of the result, the number of iterations increases dramatically with the increase in the number of DFG nodes.

To this end, we suggest to represent a large algorithm not as a flat DFG but as a hierarchical tree whose nodes are sub-graphs of the DFG. The leaves of the hierarchical tree will then be mapped, using SA, and placed on the PE array.

3.1 Dividing the Graph to Sub-graphs

An algorithm consists of three basic structures: arithmetic (logic), it-then-else, and loop structures. We can use these structures to represent the algorithm as an hierarchical tree. Each node in the hierarchical tree will represent a structure of the algorithm. The leaves of the hierarchical tree will be composed of structures that can not be further separated.

A conditional structure is separated into three siblings: *Then*, *Else* and *Cond*. Each one of the parts will be further separated if it contains a separable structure. A loop structure node will be decomposed into two parts: *Body* and *control*. The body can be replicated several times to improve performance [8]. The leaves of the hierarchical tree are arithmetic/logic structures that can not be further separated.

3.2 Mapping the hierarchical tree

The mapping process follows basically a bottom up approach. It starts from the leaves, maps them side by side and climbs up the tree until it reaches the root. The algorithm for mapping, using the hierarchical approach, is shown in Figure 6. The algorithm forms clusters in the hierarchical tree. It starts with the leaves at the lowest level and then tries to place all the nodes of the sub-graphs that belong to the same parent node together. Only in a later stage of the algorithm, leaves that are not connected to the same parent node in the hierarchy tree will be placed.

Each stage of the algorithm starts by classifying the leaf nodes at the lowest level of the hierarchy tree into distinguishable sets (S_i) according to the leaves' parent nodes. Then, for each set S_i it selects the node in the hierarchy tree that has the most links to other nodes and maps it using SA. This node is then removed from the set S_i and added to the set of mapped nodes, M_i. If there was more than one leaf in S_i, then the next node is chosen from the set S_i using the same criteria. This node is then mapped, but this time, in order to increase the array utilization, an additional parameter is added to the SA process, namely the needed aspect ratio for the resulting mapping. The aspect ratio is determined according to the nodes that had been already mapped. All the links that connect the already mapped nodes in M_i to S_i are inspected. The side where

```
l:=lowest level of the hierarchical tree
repeat
    for i:=1 to k      {k is the number of nodes in level (l-1)}
        S_i:={ all the leaves in level l of the hierarchical tree that
               are children of node_i in level (l - 1) }
    for i:=1 to k
        M_i := ∅
    for i:=1 to k
      begin
        repeat
            choose a leaf from S_i with the highest connectivity, s
            map s onto the hexagonal array
            S_i := S_i ∩ s
            M_i := M_i ∩ s
        until (S_i = ∅)
        for i:=1 to k
            compact mappings in M_i
      end
    l := l - 1
until (l = 0)      reach the root
```

Figure 6: The algorithm for mapping large algorithms.

the majority of them lie, is chosen to be the side where the new node will be placed, termed ***chosen side***. The SA will get this parameter and attempt to achieve the desired aspect ratio. This can be done by slightly changing the area weight function by adding a term which is a function of how far is the current aspect ratio from the desired one. After the leaf had been mapped, it is added to M_i. This process continues until S_i is empty (i.e., all the leaves of a mutual parent have been mapped). Then, a compaction phase is applied to the mapped nodes of M_i. The compaction phase tries to reduce the unutilized PEs in the mapped array, (e.g., tries to reduce the size of the rectangular bounded by the first and the last rows of the mapped algorithm and the first and the last columns of it). The compaction phase goes around the rectangular surrounding the mapped algorithm and tries to migrate DFG nodes from one PE to a neighbor PE in the compaction directions, to reduce the rectangular size. This process is performed on all the S_i sets at the lowest level. Then, each set M_i replaces the parent node of the leaves that have been added to it and becomes a leaf at level $(l-1)$. In the next iteration, only the leaves that were not mapped earlier will be mapped using SA. The others will be placed in the array without any modification (only the compaction phase will be performed on them). The hierarchical mapping terminates when the root of the hierarchy tree is reached (i.e., $l = 0$).

3.3 Example

We demonstrate the way large algorithms are mapped through the example in Figure 7. The outer multiply node is replicated 3 times, as well as the body part, to improve the performance [10].

The algorithm is divided into four major parts: the body replicated 3 times and the control part. Each body part is further divided hierarchically until the leaves' level is reached. Since the number of DFG nodes in the leaves is small, the whole body part has been chosen as a leaf of the tree.

Since all the body parts are identical and do not feed data one to each other, they were placed one besides the other and were combined together to create the complete mapping of the given algorithm.

```
sum:=0
for i=1 to 1000
  if a=b
    then if c=d
           then r= g*(h[i]+q[i])
           else r=j[i]+k[i]
           endif
    else r=e[i]*f[i]
  endif
  sum:=sum+r
endfor
```

Figure 7: A nested if–then–else in a loop structure.

The compaction algorithm was then applied to the resulting mapping and by migrating nodes from one PE to another, the total area of the mapping has been reduced. The final mapping is shown in Figure 8. The latency of the mapped algorithm is 68 clock units and the pipeline period is 4 which are not far from the estimations (60 and 3, respectively).

4 Conclusions

A simulated annealing algorithm has been developed to map a given DFG onto an hexagonal homogeneous data driven array. It has been shown that it is efficient for small to moderate size algorithms. Another method has been presented for dealing with mapping of large graphs.

The above mapping method assumed an unbounded array. This is not a realistic assumption. Therefore, there is a need to solve the problem of assigning DFG nodes to PEs that are on several chips. The connections available among chips have to be provided besides the PE characteristics. Once those are given, one can map the algorithm onto an array consisting of several chips using a method similar to the one presented above with additional restrictions.

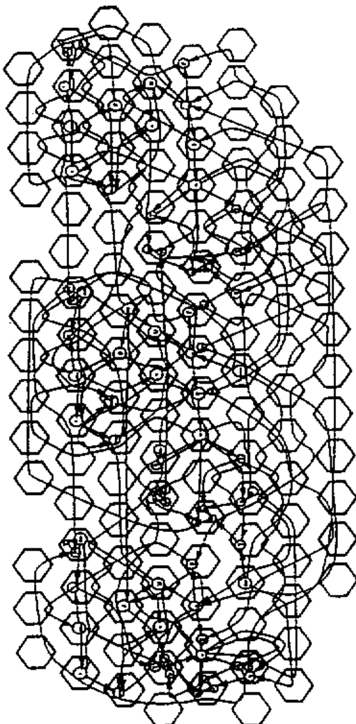

Figure 8: The mapping of the example in Figure 7 after compaction.

References

[1] R. Conrad and I. Koren. MPPE: A multiple port processing element for data-driven arrays. Tech. Rep. TR-90-CSE-4, University of Massachusetts, Amherst, 1990.

[2] David S. Johnson et al. Optimization by simulated annealing: An experimental evaluation, part I. *Oper. Res.*, 37:865–892, Nov.-Dec. 1989.

[3] A.L. Fisher, H.T. Kung, et al. Design of the PSC: A programmable systolic chip. In *Proc. Third Caltech Conf. on VLSI*, pp. 287–302, March 1983.

[4] J.A.B. Fortes and B.W. Wah. Special issue on systolic arrays–from concept to implementation. *IEEE Computer*, 20(7), July 1987.

[5] I. Koren, B. Mendelson, I. Peled, and G. M. Silberman. A data-driven VLSI array for arbitrary algorithms. *IEEE Computer*, pp. 30–43, October 1988.

[6] S.Y. Kung et al. Wavefront array processors - concept to implementation. *IEEE Computer*, 20(7):18–33, July 1987.

[7] P.J.M. Van Laarhoven and E.H.L Aats. *Simulated Annealing: Theory & Applications.* D.Reidel Publishing Company, 1987.

[8] B. Mendelson and I. Koren. Estimating the potential parallelism and pipelining of algorithms for data flow machines. Tech. Rep. TR-90-CSE-5, University of Massachusetts, Amherst, 1990.

[9] B. Mendelson, B. Patel, and I. Koren. *Designing Special-purpose Co-processors Using the Data Flow Paradigm* in *Advanced Topics in Data Flow Computing*, chapter 21, pp. 547–570. Prentice-Hall, 1991.

[10] B. Mendlson. *Mapping of Algorithms onto Programmalbe Data Driven Arrays.* PhD thesis, ECE Dept. University of Massachusetts, Amherst, 1990.

[11] S. Weiss, I. Spillinger, and G. Silberman. Techniques for mapping data flow programs onto vlsi processing arrays. In *Proc. Int'l Conf. on Fundamental Programming Languages and Computer Architecture*, 1989.

[12] D.F. Wong, H.W. Leong, and C.L. Liu. *Simulated Annealing for VLSI Design.* Kluwer Academic Publishers, 1988.