Designing Systolic Architecture For Symmetrizing Hessenberg Matrices*

Ladan Kazerouni†
Department of Mathematics
University of Bombay
Bombay, INDIA

Basant Rajan and R.K. Shyamasundar‡
Computer Science Group
Tata Institute of Fundamental Research
Bombay 400 005, INDIA

Abstract

In this paper, we describe a systematic method for mapping the problem of symmetrizing Hessenberg matrices onto systolic architectures. The starting point of our method is a graphical abstraction of the system of linear recurrence equations that specifies the problem. Using a procedure called *cubization*, we transform the dependency graph of the problem into a graph we term the *Modified Dependency Graph*. Using the modified dependency graph, we design systolic programs on a very general architecture referred to as a *basic systolic architecture*. Next, we map the basic systolic architecture onto a given systolic architecture using a set of semantics-preserving transformations and hence, the correctness of the solution comes for free. Finally, we compare the architectures for symmetrizing Hessenberg matrices obtained by our method with those proposed previously.

1 Introduction

The aim of this paper is to describe a systematic approach for mapping the problem of symmetrizing Hessenberg matrices onto systolic architectures. In

*Address to whom correspondence may be made:
†E-mail address : ladan@tcs.tifr.res.in
‡E-mail address : basant @tcs.tifr.res.in and shyam@tcs.tifr.res.in

the past, several methods for synthesizing systolic arrays have appeared in the literature [1, 2, 3, 7, 8, 9, 10]. The approach used in this paper is based on the underlying technique described in [6]. In [6], we gave a methodology to map a system of linear recurrence equations onto systolic architectures. Hence, this problem should be expressed as a system of linear recurrence equations so that the methodology in [6] could be applied.

The symmetrizer of an asymmetric matrix A is a symmetric matrix X such that $XA = A^tX$, where A^t is the transpose of the matrix A. A symmetrizer is used to convert an asymmetric matrix into an equivalent symmetric matrix whose eigenvalues are the same as those in the asymmetric matrix and hence, is useful in several engineering applications.

Let $B = [b_{ij}]$ be a lower Hessenberg matrix with $b_{ii+1} \neq 0$ where $i = 1 \ldots n$ and let x_i be the i^{th} row of the symmetrizer X. From $XB = B^tX$, the sequential algorithm is given below:
`STEP 1`: Choose any x_i where $x_i \neq 0$. Let $x_n = x_i$.
`STEP 2`: Compute $x_{n-1}, \ldots, x_1$ recursively from:

$$x_i = \frac{1}{b_{i,i+1}}(x_{i+1} * B - \sum_{l=i+1}^{n} b_{l,i+1}x_l) \quad (1)$$

where $B = [b_{ij}]$ is the lower Hessenberg matrix.

Definition 1 (**Linear recurrence equation**) Let L_n denote the lattice points in Euclidean n-space E_n. Any point $p \in L_n$ can be designated by an n-dimensional vector whose coordinates are integers. Linear recurrence equations (LRE) are equations of the form:

$$\begin{aligned} a_k(p_0) &= m_k, \\ a_k(p) &= f_k(a_1(\delta_1(p)) \ldots a_s(\delta_s(p))) \ , \ k \in \{1 \ldots s\} \end{aligned} \quad (2)$$

where

1. a_i is a function from L_m to N,
2. δ_i is an affine function from L_n to L_m such that $\delta_i(p) = A_ip + b_i$,
   - A_i is a constant $m \times n$ matrix,
   - b_i is a constant $m \times 1$ matrix.
3. $a_k(p)$ does not appear on the r.h.s of the equation,
4. m_k is a scalar constant (input) and

5. f_k is a single-value function strictly dependent on its arguments.

Our first task is to bring equation 1 into the LRE format given above. For this purpose x and b are to be in Z^3 and the iterative equation is rewritten using tail recursion.
The basic idea is to introduce an auxilary variable for storing the partial sums. Thus, the initialization of x,b and h lead to :

- normalizing the indices and assigning initial values of x and b

$$x(i,j,k) = x_{nj}, \quad \text{for } i = n, k = n$$
$$b(i,j,k) = b_{ij}, \quad \text{for } k = 0, 1 \leq i \leq n, 1 \leq j \leq n$$

- introducing h as a partial sum for computing the second factor of equation (1) and setting it to 0 ...

$$h(i,j,k) = 0, \quad \text{for } k = 0, 1 \leq i < n, 1 \leq j \leq n$$

Now, as per equation (1) we can define h in two parts :

- computing the 1^{st} term of the 2^{nd} factor of equation (1) for $k \leq i$ ...

$$h(i,j,k) = h(i,j,k-1) + x(i+1,k,n) * b(k,j,0), \quad \text{for } 1 \leq i < n, 1 \leq j \leq n, 1 \leq k \leq i \quad \text{(a)}$$

- computing the 1^{st} and 2^{nd} terms of the 2^{nd} factor of equation (1) for $k > i$ ...

$$h(i,j,k) = h(i,j,k-1) + x(i+1,k,n) * b(k,j,0) - b(k,i+1,0) * x(k,j,n), \quad \text{for } 1 \leq i < n, 1 \leq j \leq n, i < k \leq n \quad \text{(b)}$$

Finally, relating x, h and b leads us to the following :

- dividing the 2^{nd} factor of equation 1 by its 1^{st} factor ...

$$x(i,j,k) = h(i,j,n)/b(i,i+1,0). \quad \text{(c)}$$

Rewriting equations (a),(b),(c) in the form of equation (2) we get :

$$h(p) = f_{h_1}(h(\delta_1(p)), x(\delta_2(p)), b(\delta_3(p))) \quad \text{(for the case where } k \leq i\text{)}$$
$$h(p) = f_{h_2}(h(\delta_1(p)), x(\delta_2(p)), b(\delta_3(p)), b(\delta_4(p)), x(\delta_5(p))) \quad \text{(for the case where } k > i\text{)}$$
$$x(p) = f_x(h(\delta_6(p)), b(\delta_7(p)))$$

- where $p = (i,j,k)$,
- $f_{h_1}(x,y,z) = x + y * z$,
- $f_{h_2}(x,y,z,x',y') = x + y * z - x' * y'$,
- $f_x(a,b) = a/b$ and

$$\delta_1(i,j,k) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * p + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = (i,j,k-1)$$

$$\delta_2(i,j,k) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} * p + \begin{bmatrix} 1 \\ 0 \\ n \end{bmatrix} = (i+1, k, n)$$

$$\delta_3(i,j,k) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} * p = (k, j, 0)$$

$$\delta_4(i,j,k) = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} * p + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = (k, i+1, 0)$$

$$\delta_5(i,j,k) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} * p + \begin{bmatrix} 0 \\ 0 \\ n \end{bmatrix} = (k, j, n)$$

$$\delta_6(i,j,k) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * p = (i, j, n)$$

$$\delta_7(i,j,k) = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} * p + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = (i, i+1, 0)$$

Note: The symmetrizing Hessenberg matrix problem is different from the problem addressed in [6]. In [6], the vector b_i is a constant independent of the problem size whereas here, the vector $[1,0,n]^t$ and $[0,0,n]^t$ are constant only for a given problem. However, we show that even in such cases our method is applicable.

2 Designing a Systolic Program for Symmetrizing Hessenberg matrix

In this section, we describe our method briefly [5, 6]. Broadly our method consists of the following steps:

1. Deriving a graphical abstraction, referred to as the dependency graph, for the given LRE.

2. Transforming the dependency graph using the cubization procedure to obtain the timing function for the LRE.

3. Constructing the BSA.

4. Mapping BSA to a given architecture.

Using equation 1 we construct the dependency graph for symmetrizing 3×3 Hessenberg matrix.

The dependency graph of symmetrizing Hessenberg matrices is shown in Figure (1).

2.1 Deriving a Timing function

We now apply a procedure called *cubization* to transform the DG into a *modified dependency graph* [6]. The purpose of this procedure is to identify the functions whose computation can be carried out independently and without interference. In other words, two children of the same node with the same label will not be computed at the same time. Such a condition guarantees that one does not require *global memory*. On the other hand, to ensure that parallelism is not precluded, we constrain the ordering to those children that belong to the same child-set. (*Please refer to definitions direct dependence, dependence and child-set* [6]). This ordering is defined using a set of *rewrite rules*. Note that the new ordering preserves the ordering imposed by the dependency graph. The application of these rules, help to arrive at a data flow pattern that the systolic array will eventually implement.

The application of the cubization procedure eventually generates what we call the *modified dependency graph*. This graph is a forest wherein each path starting from a root represents the flow of a particular datum. The relative position of a node on a chain imposes an execution ordering. This preserves the dependencies imposed by the dependency graph, but is further refined. The level at which a given vertex occurs in a chain represents its associated timing function and this is *consistent* because a particular vertex occurs at the same level in all chains, if it occurs at all. The structure of the modified dependency graph obviously precludes the use of a particular datum by more than one node at the same time, thus *linearising* the dependencies. The association of the depth of a vertex in the modified dependency graph to its timing function is modelled on the *free schedule* of *Karp et al* [4]. In short, our transformation consist of following steps:

Steps of the Transformation:
I. **Preprocessing Steps**

1. All children of a given node, are partitioned into classes of child-sets.

2. The edges leading into classes of child-sets are relabeled such that all edges leading to child nodes in the same partition are identical, while those leading to child nodes in different partitions are distinct.

II. **The cubization procedure**

1. recursively divides the elements of child-sets into clusters,

2. defines an order among these clusters and connects them by labeled edges.

The MDG corresponding to DG of symmetrizing Hessenberg matrices is shown in Figure (2).

2.2 Basic Systolic Architecture (BSA)

The MDG is a set of linear chains of computations. The labels repeated (in different chains) correspond to the same computation. To obtain a systolic architecture, we have to embed all these chains into a lattice structure. The process requires identifying the sub-sequences of chains that can be overlaid on other chains.

After embedding these chains into a lattice structure we elminate non-local communication with the help of two procedures called *folding* and *staggering* [5].

- **Folding:** The purpose of folding chains along the coordinate axes chosen is to see that links need only be made to adjacent nodes. The procedure ensures that any non-local communication that the algorithm required is re-routed through alternate paths.

- **Staggering:** The folding of the non-parallel chains leads to multiple edges between lattice nodes. Hence, the timing function has to be refined so that proper data is sent to the proper node at the proper time.

The BSA for symmetrizing Hessenberg matrices is shown in Figure (3).

2.3 Mapping

A mapping from BSA to another network G' is a triple $\mathcal{M}= (e, d, g)$ where,
• e is a projection scheme. By a projection scheme we mean *'projection along an axis'* or *'translation along an axis'*.

• d is a function that maps a channel in BSA to a channel in G'.
• The timing function g maps the execution time associated with each computation in BSA to the execution time associated with each computation in G'.

3 Final Remarks

We have presented a mapping of an algorithm for symmetrizing a Hessenberg matrix onto a systolic array. Though the solution presented employed a four dimensional array, it could be mapped onto a two dimensional array using *projections* as mentioned above. Observation of the BSA obtained will show that the time complexity of the implementation is $O(n^2)$ with unity being the coefficient of the n^2 term. It should be noted that staggering will only add to the constant term in the expression for time complexity in this case. This compares favourably with Kung-Leiserson systolic implementation. Furthermore, the method is completely automated.

References

[1] J.M. Delsome and I.C.F Ipsen *Systolic Array Synthesis: computability and time cones*, In Parallel Algorithms and Architecture, 1986, pp. 295-312.

[2] J.A.B. Fortes and D.I. Moldovan, *Parallelism detection and transformation techniques for VLSI algorithms*, Journal of Parallel and Distributed Computing, **2**, 1985, pp. 277-301.

[3] H.V Jagadish, Sailesh K. Rao and Thomas Kailath *Array Architecture for Iterative Algorithms*, Proc. IEEE, VOL. 75, NO. 9, pp 1304-1321, September 1987.

[4] R.M Karp, E.M Miller and S. Winograd, *The Organization of Computations for Uniform Recurrence Equations*, J. of the Association for Computer Machinery **14**, 1967, pp. 503-590.

[5] L. Kazerouni , B. Rajan, R.K. Shyamasundar *Derivation of Systolic Programs*, Proc. IEEE International Conference on Parallel Processing, 1994.

[6] L. Kazerouni , B. Rajan, R.K. Shyamasundar *Mapping Linear Recurrences onto Systolic Arrays*, Technical Report 3/94 TIFR.

[7] P. Quinton, *Automatic synthesis of systolic arrays from uniform recurrence equations*, Proc. IEEE 11th International Symposium on Computer Architecture, 1984, pp. 208-214.

[8] P. Quinton and V.V. Dongen, *The Mapping of the Linear Recurrence Equations on Regular Arrays*, Technical Report No. 1093, INRIA, France, 1989.

[9] S.Rajopadhye, S. Purushhothaman and Richard M. Fujimoto, *On Synthesisizing Systolic arrays from Recurrence Equations With Linear Dependencies* Proc, 6th FSTTCS, India, December 1986. LNCS NO. 241, Springers.

[10] S. Rajopadhye, *Synthesizing systolic arrays with control signals from recurrence equations*, Dist. Computing **3**, 1989, pp. 88-105.

Appendix

Figures referred to in the paper are indicated below.

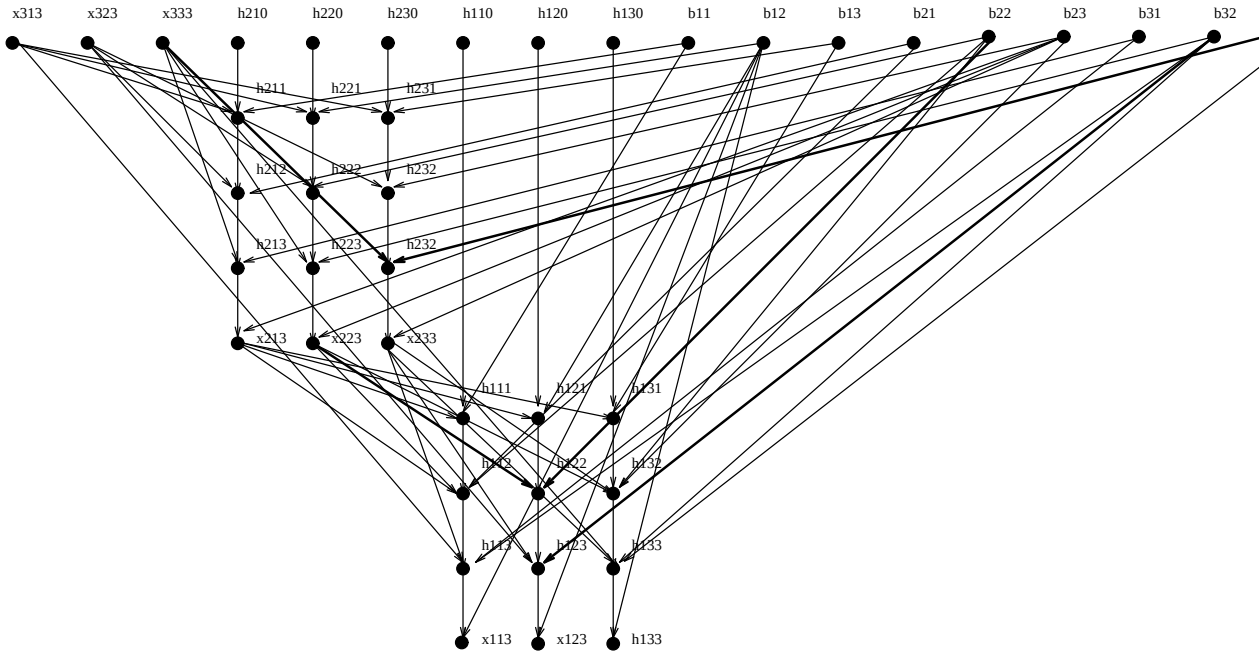

Figure 1: Dependency graph for symmetrizing 3×3 Hessenberg matrix

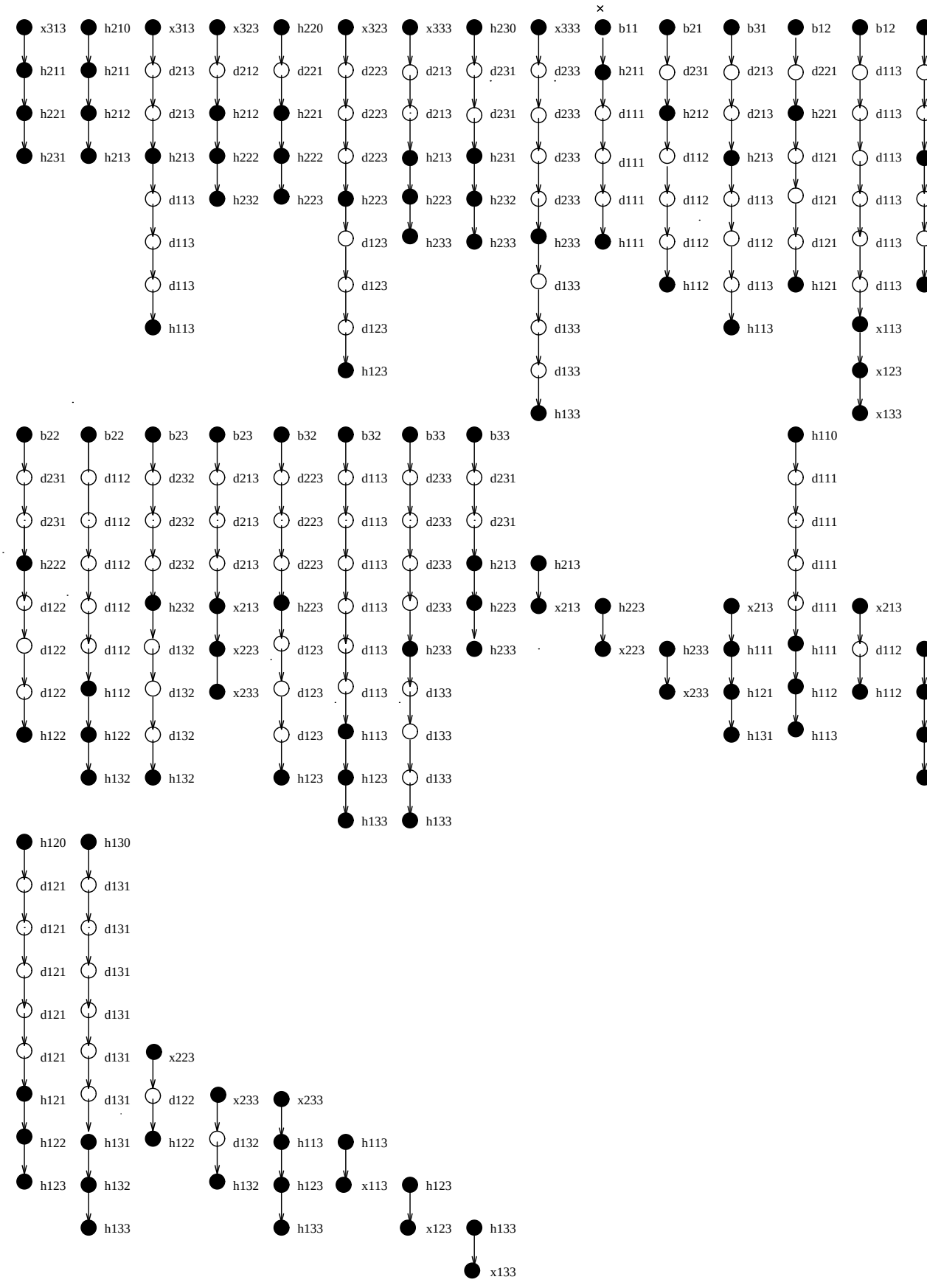

Figure 2: The MDG for symmetrizing 3×3 Hessenberg matrix

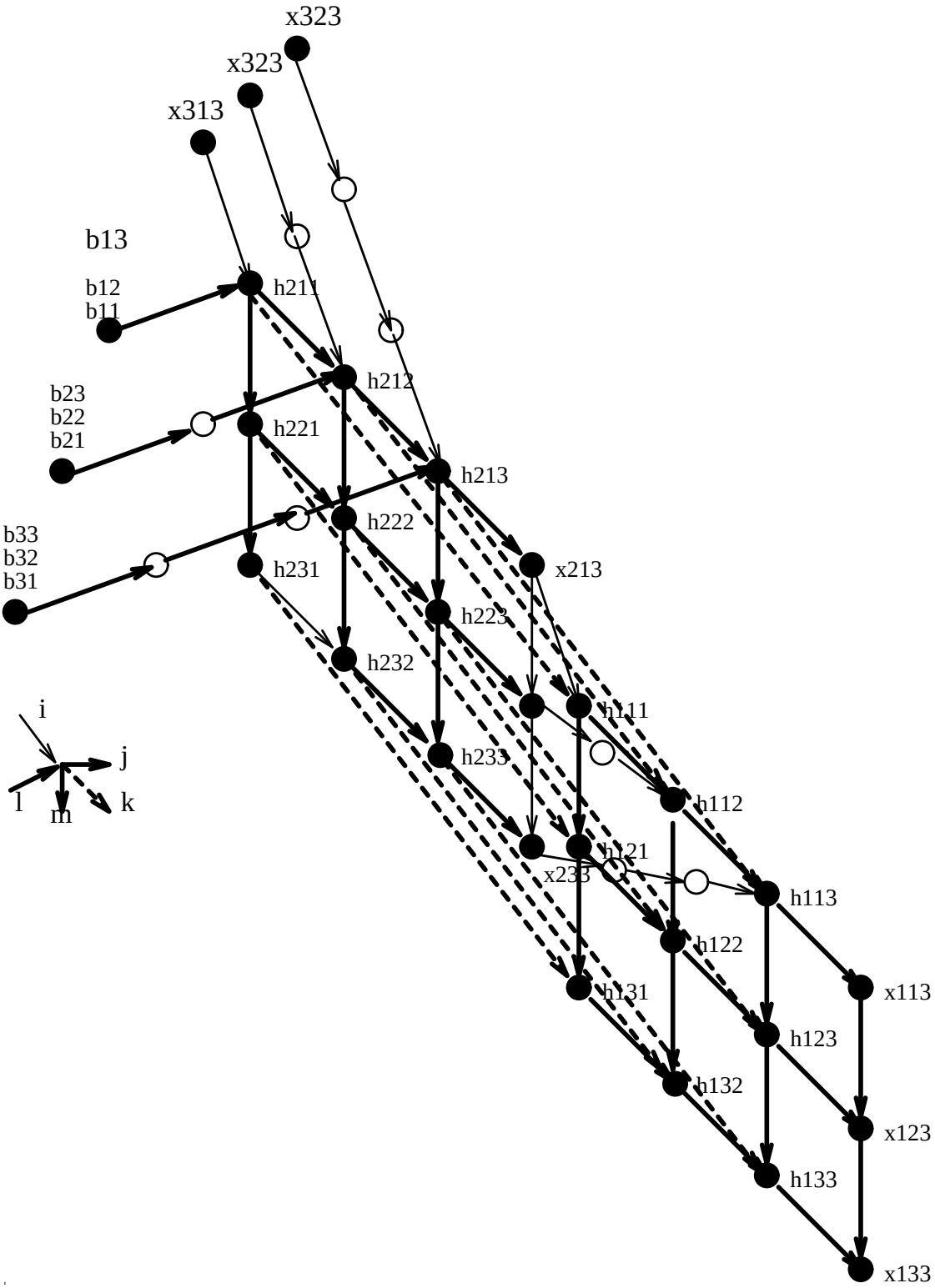

Figure 3: The BSA for symmetrizing 3×3 Hessenberg matrix