Rewiring - A New Design Technique for Systolic Systems
(Extended Abstract)

by

A. Litman

Technical Report #740
June 1992
Rewiring - a New Design Technique for Systolic Systems

(extended abstract)

Ami Litman
Dept. of Computer Science, Technion, Haifa, Israel
litman@cc.technion.ac.il

February 18, 1992

Abstract

Rewiring is a technique that changes the interconnection of a system in such a way that the system keeps performing exactly the same computation, but in different places. This paper presents a wide range of applications of this technique: transformation of space into time, unrolling a computation, optimizing a k-slow system, trading speed for hardware, efficient multiplication of band matrices, eliminating busses from a mesh, and more. Although most of the individual results above are known, this paper reveals that there is a single fundamental technique behind all of them. We expect this technique to become a valuable tool for designing systolic systems.

1 Introduction

Rewiring is a technique that changes the interconnection of a system in such a way that the system keeps performing exactly the same computation, but in different places. This paper presents a wide range of applications of this technique: transformation of space into time, unrolling a computation, optimizing a k-slow system, trading speed for hardware, efficient multiplication of band matrices, eliminating busses from a mesh, and more. Although most of the individual results above are known, this paper reveals that there is a single fundamental technique behind all of them.

We assume that the reader is familiar with the basic terminology of systolic systems. We use here the terminology of [8]. In general, the word “system” means a semisystolic system with several (including zero) hosts. In most cases, it is useful to consider each external edge as having its own host.

2 Rewiring

Let \( Y \) be a semisystolic system with a communication graph \( G = (V, E, w) \). We say that \( \pi \) is an automorphism of \( Y \) if the following conditions hold.

- \( \pi \) is a function from \( V \cup E \) into \( V \cup E \).
- \( \pi \) is an automorphism of the graph \((V, E)\).
- \( \pi \) preserves the weights of the edges: \( w(\pi(e)) = w(e) \).
- \( \pi \) maps hosts to hosts and combinational units to combinational units.
- \( \pi \) preserves the functionality of the combinational units. If \( \pi(v) = u \) where \( v \) and \( u \) are combinational units, then \( v \) and \( u \) compute the same function and \( \pi \) maps the edges incident upon \( v \) to the corresponding edges of \( u \) in a “proper” manner. For example, if \( v \) has two input ports \( a \) and \( b \) and it computes the (nonsymmetric) function \( a - b \), then \( \pi \) maps the edge entering port \( a \) of \( v \) to the edge entering port \( a \), rather than port \( b \), of \( u \).

Given an automorphism \( \pi \), we extend it to be a permutation of the registers of \( Y \) as follows. Let \( r \) be a register on an edge \( e \), then \( \pi(r) \) is the corresponding register on the edge \( \pi(e) \). We also extend it to be a permutation of the states of \( Y \). For any \( s \), a state of \( Y \), \( \pi(s) \) denotes the state generated from \( s \) when we move the content of every register \( r \) into \( \pi(r) \). Assume \( Y \) has no hosts, and let \( N \) denote the next-state function of \( Y \). I.e., \( Y \) moves from state \( s \) to state \( N(s) \) in one tick. It is easy to verify that \( N \) and \( \pi \) commute. That is, for any state \( s \),

\[
\pi(N(s)) = N(\pi(s)) \tag{1}
\]

Let \( Y \) be a semisystolic system with an automorphism \( \pi \). Consider the following “system” \( Y^* \). It has the same
there are two output ports,

four identical combinational units connected as a ring.

Each combinational unit has three input ports. They

matrix-vector product

and since

is, \( Y' \) is a semisystolic system. Similarly, if \( Y \) is systolic,

2.1 An Example

Figure 1(a) shows a systolic system

Figure 1: (a) Systolic system \( Z \) computing \( y = Ax \), and
(b) its rewiring \( Z^* \).

rewiring we need to know, not only the nodes incident
to each edge, but also the edges the edge is connected to.) The combinational unit computes \( z_{\text{out}} = z_{\text{in}} \) and \( y_{\text{out}} = y_{\text{in}} + a_{\text{in}} z_{\text{in}} \).

In this figure, small bars on the edges denote registers,
and the values written next to them describe the initial
state of the system. The vector \( z \) is already in place,
and the registers that hold the intermediate values of \( y \)
are cleared. (We ignore the issue of how these values
got there.) The \( a \) values above the system denote the
elements of the matrix \( A \) to be provided by the host.

System \( Z \) computes \( y = Ax \) in four cycles. During
the computation, the \( y \) values are stationary, while the \( z \)
values travel rightward in a circular fashion. Assume we
wish to transform the system (and the computation) in
such a way that the \( z \) values will be stationary and the \( y \)
values will travel. This can be done by rewiring as follow.
Let \( \pi \) be the automorphism that maps each node to its
left neighbor (with wraparound at the leftmost node).
Figure 1(b) describes the system \( Z^* \). Note that the edges
carrying the \( z \) values become self-loop, while the edges
carrying the \( y \)'s are going leftward. Also note that the
input array is modified. If one imagines the \( a \)'s to reside
on a cylinder, this cylinder is twisted.

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix} \]

\[
\begin{cases}
y_1 = 0, & y_2 = 0, & y_3 = 0, & y_4 = 0 \\
z_1, & z_2, & z_3, & z_4
\end{cases}
\]
2.2 Rewiring and retiming

There is a profound similarity between rewiring and retiming ([9], [10], [3]). Both techniques transform a semisystolic system into a new system with similar functionality, and both keep the combinational units intact. However, rewiring and retiming differ in several ways.

- Rewiring changes the interconnection while retiming does not. It only changes the number of registers on the edges.
- To apply rewiring, we need to know something on the functionality of the combinational unit for finding useful automorphisms. In contrast, retiming is purely syntactic — the functionality of the combinational units is irrelevant.
- Rewiring does not change the Input/Output timing, but changes their location. On the other hand, retiming does not change the location of the Input/Output, but may change their timing in the case of multiple hosts.

The relationship between rewiring and retiming reveals the duality of space and time in systolic computations. The former transforms the space while the latter transforms the time. These two techniques complement each other, and we shall see examples where it is useful to apply them both, one after the other, on the same system.

3 Transforming Space into Time

To demonstrate the duality of space and time, we describe here a technique that, under certain conditions, transforms a system and its computation in such a way that a dimension of space is mapped into time. That is, for any node $z'$ of the new system $Y'$, there is a sequence $(z_1, z_2, \ldots, z_k)$ of distinct nodes of the original system $Y$, and an instant $t$, such that the same computation performed by $(z_1, z_2, \ldots, z_k)$ at time $t$ (in $Y$) is performed by $z'$ (in $Y'$) in a time interval $(j + 1, j + 2, \ldots, j + k)$ for some $j$. (Since our systems are finite, we can only map a finite section of space, like $(z_1, z_2, \ldots, z_k)$ above, into time.) Such a transformation can be achieved by applying both retiming and rewiring, as follows.

Assume we can partition the nodes of a system into $k$ disjoint sets, $\{S_i \mid i \in \{1, k\}\}$, such that there is an automorphism $\pi$ satisfying $\pi(S_i) = S_{i-1}$ for every $i$. Unfortunately, this can never happen in a finite system — $S_1$ has no place to go, and mapping it into $S_k$ will not do. Let us try again. Assume $Y$ is a (finite) system that can be extended into an infinite system $Y_1$, and $Y_1$ can be partitioned into disjoint sets, $\{S_i \mid i \in Z\}$, such that $\bigcup_{i=1}^{\infty} S_i$ contains exactly the nodes of $Y$, and there is an automorphism $\pi$ satisfying $\pi(S_i) = S_{i+1}$ for every $i$. Let $r(v)$ be the index of the partition having node $v$. (I.e., $v \in S_{r(v)}$ for every $v$.) Finally, assume

$$w_1(e) \geq r(tail(e)) - r(head(e))$$

for any edge $e$ of $Y_1$. Under these conditions, we can perform the requested transformation. However, before doing that, let us check how restrictive these conditions are. Define an edge to be a backward edge if it goes from some $S_i$ into $S_{i-j}$ where $j$ is strictly positive. Condition 3 demands that we have "enough" registers on backward edges. It does not impose any restriction on interpartition or "forward" edges. There are two popular architectures of fixed-connection networks that may satisfy the conditions above — the butterfly and the mesh [8]. The number of levels of the butterfly or the number of column (say) of the mesh have to be a multiple of $k$. Also all the combinational units have to be identical. If the latter condition does not hold, we can "fix" it by homogenizing the system, as will be discussed in the next section.

Transform the computation of $Y_1$, (and hence of $Y$) as follows.

- Transform $Y_1$ into $Y_2$ by retiming it with the function $r - 1$. This operation causes any node $v$ to lag $r(v) - 1$ cycles by removing $j$ registers from any edge going from $S_i$ to $S_{i-j}$. By inequality 3 we have enough registers to perform this retiming.
- Transform $Y_2$ into $Y_3$ by rewiring it with the automorphism $\pi$. (Note that $\pi$ is an automorphism of both $Y_1$ and $Y_2$.)

As an example, figure 2 illustrates these transformations on a system $W$ computing $y = Ax$ where $A$ is a $4 \times 3$ matrix and $x$ and $y$ are vectors of three and four elements respectively. The "backward" direction in this figure is from left to right.

To find the effect of these transformation on the computation, define $\nu_1(v, t)$ to be the output of node $v$ at cycle $t$ in the system $Y_1$, and let $\nu_2$ and $\nu_3$ be the corresponding notations for systems $Y_2$ and $Y_3$. These $\nu_i$ depend implicitly on the inputs provided by the corresponding hosts, and we assume that the hosts properly transform their input. The $\nu_i$ should also depend on the initial state of the systems, but we overcome this by assuming that the systems were always running. Since $Y_2$ is a retiming of $Y_1$ by $r - 1$,

$$\nu_2(v, t) = \nu_1(v, t - r(v) + 1)$$


for any node $v$ and cycle $t$. By abuse of notation, let $v+i$ stand for $x_i^{(i)}(v)$. Since $Y_3$ is a rewiring of $Y_2$ by $r$, we have

$$\nu_3(v,t) = \nu_2(v-t,t)$$

By combining these equations, and noting that $r(v-t) = r(v) + t$, we get

$$\nu_3(v,t) = \nu_2(v-t,t) = \nu_1(v-t, t - r(v-t) + 1) = \nu_1(v-t, -r(v) + 1)$$

Assume it is sufficient to simulate the computation performed by $Y$ during the finite time interval $[1, m]$. In this case, we can truncate $Y_3$, keeping partitions $S_0$ thru $S_{m-1}$. Let $Y_4$ be the resulting finite system. (See figure 2(d).)

The transformation from $Y$ to $Y_4$ induces a transformation of space into time, but not of time into space! The sequence $\{\nu(v, 1), \nu(v, 2) \cdot \cdot \cdot \}$ from $Y$ computation is mapped to $\{\nu_4(u, j), \nu_4(u + 1, j + 1) \cdot \cdot \cdot \}$ for some $u$ and $j$. To transform space into time and time into space simultaneously, system $Y_1$ has to satisfy a rather strong condition:

- $Y_1$ has no backward edges.

This is a necessary and sufficient condition to enable a retiming of $Y_4$ by $r$. Let $Y_5$ be the resulting system. (See figure 2(e).) Note that if $e$ is an edge of $Y_1$ that goes from $S_i$ to $S_{i+j}$ and has $g$ registers, then $e^r$ of $Y_5$ goes from $S_i$ to $S_{i-x}$ and has $j$ registers. I.e., the combined transformation transposes distances with registers, and this is why we cannot have backward edges.

Since $Y_5$ is a retiming of $Y_4$ by $r$:

$$\nu_5(v,t) = \nu_4(v,t-r(v)) = \nu_1(v-t+r(v), -r(v)+1)$$

and note that $v+r(v)$ is always a member of $S_0$. Hence, the computation of $Y_5$ during the time interval $[1, k]$ is an "exact" translation of the computation of $Y$ during the interval $[1, m]$.

The conditions under which we may transform $Y$ to $Y_5$ (or $Y_4$) depend on $k$ and we do not know any feasible algorithm to find out for a given $Y$ and $k$ whether such a construction is possible. It seems that this problem is related to the graph isomorphism problem [4].

The construction of $Y_5$ from $Y$ is always possible in the special case of $k = 1$. The $S_i$ are unconnected copies of $Y$ and all the conditions trivially hold. In this case, $Y_5$ is an $m$-unrolling of $Y$. That is, $Y_5$ is an acyclic network without registers, and computes (in a single cycle) what $Y$ computes in $m$ cycles. For example, if $Y$ is a serial adder then $Y_5$ is an $m$-bit ripple adder.

---

![Figure 2: Transforming space into time. (a) System $W$ computing $y = Ax$. (b) $W_2$ - an infinite extension of $W$ and a retiming by $r-1$. (c) $W_3$ - a rewiring of $W_2$ by $r$. (d) $W_4$ - a finite fraction of $W_3$. (e) $W_5$ - a retiming of $W_4$ by $r$.](image-url)
4 Homogenizing a System

In some cases, diversity of the combinational units eliminates useful automorphism of a given system. We can repair this by adding hardware as follows. Call $\pi$ an almost-automorphism if it is an automorphism of the communication graph, but not necessarily preserves the functionality of the combinational units. We say that a semisystolic system is homogeneous if all its almost-automorphisms are automorphisms. We claim that any semisystolic system $Y$ can be "patched" to become homogeneous. The details of this construction are omitted here, but the main idea is to replace the combinational units of $Y$ with "universal combinational units". For each $i$ and $j$ let us have a unit that can simulate all the combinational units of $Y$ having $i$ and $j$ input and output ports. We need, somehow, to tell each universal unit which actual unit to simulate. We will not describe this here. Usually, a unit that can simulate several other units has the hardware of all of them combined. In general, a single universal unit can be as large as all the original system. However, "honest" systolic algorithms utilize only a fixed number of types of combinational units, regardless of the size of the system. In this case, our construction increases the "size" of the system by only a constant factor.

5 Untying a $k$-slow System

A $k$-slow system ([9], [10]) is one that runs $k$ independent sub-computations (called calculations) simultaneously by time-multiplexing all the combinational units. I.e., each node works in one cycle on the first calculation, the next cycle on the second calculation, etc. There is no flow of information between these calculations. Under certain conditions we may rewire a $k$-slow system $Y$ into a system $\tilde{Y}$, such that $\tilde{Y}$ is composed of $k$ unconnected systems, $\tilde{Y}_1, \tilde{Y}_2, \ldots, \tilde{Y}_k$, all isomorphic, and each $\tilde{Y}_i$ runs a single calculation. Metaphorically, we untie the knot $Y$ into $k$ disjoint knots. Since rewiring changes neither the number of registers, nor the combinational units, the "size" of a single $\tilde{Y}_i$ is $1/k$ of the size of $Y$. If all we need is a single calculation, we can use a single $\tilde{Y}_i$ — a great saving in hardware.

Let $Y$ be a $k$-slow system. Then there is a partitioning of its nodes $\{ S_i | i \in [0, k-1] \}$ such that for any edge $e$ going from $S_i$ to $S_j$:

$$i + w(e) \equiv j \pmod{k}$$

Assume there is an automorphism $\pi$ satisfying $\pi(S_i) = S_{i+1}$ for all $i$ (where $'-'$ is mod $k$). Then $\tilde{Y} = Y^*$ is the requested system and $\tilde{Y}_i = S_i$.

To verify that the $S_i$ are unconnected in $\tilde{Y}$, consider an edge $e$ of $Y$ going from $S_i$ to $S_j$. In $\tilde{Y}$, $e^*$ goes from $S_{i+1-w(e)}$ to $S_j$.

5.1 An Example

Figure 3: (a) System $Y'$ computing the convolution of $<x_1, x_2, x_3>$ and $<y_1, y_2, y_3>$. (b) $Y - Y'$ extended and rewired. (c) $\tilde{Y}$ — the active section of $\tilde{Y}$.

Figure 3(a) describes a 2-slow systolic system $Y'$ (from 1.2.4 of [8]) computing the convolution of $<x_1, x_2, x_3>$ and $<y_1, y_2, y_3>$. As shown, the system is somewhere in the middle of this process, and a "*" represents a value of the second calculation. The system $Y'$ does not have the required automorphism, but we can "fix" that in two ways: either extend it to an infinite system, or wrap it around. Let us choose the former method which is simpler. Let $\bar{Y}$ be the infinite extension of $Y'$, and $\pi$ be the automorphism of $\bar{Y}$ that maps each node to its right neighbor. Figure 3(b) shows $\bar{Y} = Y^*$. Note that $\bar{Y}$ has two unconnected components, and we may discard one of them. Also note that the edges carrying the $y$'s become self-loop in $\bar{Y}$. Hence, only three nodes of $\bar{Y}$ are actually working, and we may discard the rest of the system. See the resulting finite system, $\tilde{Y}$, in figure 3(c).

In addition to being half of $Y'$, the systolic system $\tilde{Y}$ has other attributes.

- If the $y$ vector is constant, there is no need to feed it in.
The output is delivered serially.

The system has only $|y|$ nodes, regardless of the size of $z$.

6 Trading Speed for Hardware

We may utilize the technique of the previous section to transform a given system into a system that is $k$ times slower, but has only $1/k$ of the original combinational logic. (There is no way to reduce the registers, however.)

If $Y$ is a $k$-slow system that can be untied, we can reduce it by a factor of $k$ without losing speed. But what shall we do if $Y$ is an "honest" system? We can make it dishonest via a slowdown by $k$, and then try to untie it. This scheme works for certain systems, including butterflies and meshes without wraparound.

Let $Y$ be a systolic system and $\{A_i \mid i \in [0, n-1]\}$ a partitioning of its nodes, where $n \equiv 0 \pmod k$ and all the inter-partition edges connect neighboring partitions (no wraparound). Assume we can add dummy edges between $A_0$ and $A_{n-1}$ such that the resulting system, $Y'$, has an automorphism $\pi$ satisfying $\pi(A_i) = A_{i+1}$ (where $'+' \equiv \text{mod} n$). In this case we can reduce the size of $Y$ as follows. Slow $Y'$ down by a factor of $k$ via substituting $k$ registers for each register of $Y'$. The resulting system, $Y''$, does not satisfy the conditions of section 5 — all the $S_i$ but one, are empty. We fix that by a makeshift retiming. Add one register to each edge going from $A_i$ to $A_{i+1}$ (including $A_{n-1}$ to $A_0$) and remove one register from each edge going in the reverse direction. Call this new system $Y'''$. If we ignore the dummy edges, the transformation of $Y''$ into $Y'''$ is a legal retiming. The systolic system $Y'''$ satisfies the conditions of section 5 and can be untied.

Note that if the original system $Y$ is a mesh, the end result is a mesh with a wraparound on one dimension. If $Y$ is a butterfly, then the end result looks like a wraparound butterfly, but is not — it has the wrong number of levels.

7 Multiplication of Band Matrixes

Kung and Leiserson [7] have invented an ingenious scheme to multiply two band matrixes on a $w_1 \times w_2$ hex-connected systolic system, where the $w_i$ are the band widths. This has been improved by Huang and Abraham [6] who constructed a system that, although almost identical to the original one, is three times faster. (The original system is 3-slow.) While it is rather easy to verify that these systems work, it is much harder to design them or similar systems, and the authors have not provide a methodology toward this end.

![Diagram of Band Matrix Multiplication](image)

Figure 4: Band matrix multiplication

Let us see how the system of Huang and Abraham can be designed in a straightforward manner via rewiring. Consider multiplication of two dense $n \times n$ matrices, $C = AB$. Figure 4(a) depicts the "standard" way to perform this computation on an $n \times n$ systolic mesh. The horizontal edges carry the $a$'s, the vertical edges carry the $b$'s, while the self-loops carry the intermediate values of the $c$'s. The $A$ and the $B$ represent the input schedule of the host, while the shaded square is the systolic mesh. But there is another way to interpret the same picture! Imagine that the mesh is infinite and occupies the whole plane. The $A$ and $B$ matrixes are already within the system — their values reside in the appropriate registers. All other registers are clear. Under this interpretation the shaded square is the active area of the mesh where the actual computation takes place. Rewire this infinite system by the automorphism that maps each node to its upper left neighbor. See figure 4(b). The shaded hexagon is the active area which is three times larger.
than the original. Hence, this system is inefficient for multiplying dense matrixes. However, consider $A$ and $B$
to be band matrixes, where the hatched areas contain the
bands. In this case, the active area is the black rectangle
which has $w_1 \times w_2$ nodes (even if the bands do not con­
tain the main diagonal). This active area is the system
of Huang and Abraham.

The system of Kung and Leiserson is identical to that
of Huang and Abraham, except that the direction of the
c edges is reversed. Their system can also be generated
from the “standard” system using rewiring, retiming and
slowdown. We will not describe that here, however.

8 The Bussing Mesh

To demonstrate the conceptual power of rewiring, we de­
fine here a nonsystolic extension of the mesh that is easier
to use than semisystolic systems. Applying rewiring, we
show that, for “normal” algorithms, this model can be
simulated by a mesh of the same size. While doing that, we
present an easy way to “program” the mesh to com­
pute transitive closure.

Consider a semisystolic system augmented by buses.
A bus, by our definition, is a device that conveys a value
between several nodes. In each cycle, exactly one of these
nodes, called the bus master, transmits a value on the
bus. This value reaches all the nodes in the next cycle.
See figure 5 for a linear array with a bus.

Figure 5: A linear array with a bus

Each node uses only local information (i.e., its current
inputs) for deciding whether to become a master. It is the
duty of the algorithm to guarantee that there is exactly
one master in each cycle.

A bussing mesh is a systolic mesh augmented by buses:
one bus for each row and one for each column. We say
that an algorithm for the bussing mesh is normal if the
following holds.

- In any cycle, all masters of the column buses are in
the same row, called the master-row.
- Each cycle, this mastership moves at most one row
up or down (with wraparound).
- The initial position of the mastership does not de­
pend on the data.

- All nodes (not only the masters) “know” which way
the mastership moves.
- The dual conditions hold for the columns.

To illustrate the strength of normal algorithms on the
bussing mesh, let us see how Dantzig’s (serial) algorithm
for transitive closure [2] is translated to this model.

The algorithm runs on an $n \times n$ bussing mesh, where
$n$ is the number of vertices in the given graph. The algo­
ритм takes $n$ cycles and the masterships move in a
single direction. In cycle $k$, the master-row and master­
column are the $k$th row and $k$th column. Initially, the
adjacency matrix of the graph resides in the mesh. Each
node $<i,j>$ holds (on its self-loop) the corresponding
binary value. Consider this bit to be a monotonic ap­
proximation of the predicate: “$i$ is connected to $j$”. In
each cycle, every row-master sends its bit to the whole
row, and every column-master sends its bit to the whole
column. Each node $<i,j>$ updates its bit according to
the rule: “if there is a path from $i$ to $k$ and a path from
$k$ to $j$, then there is a path from $i$ to $j$”. As in the serial
algorithm of Dantzig, the mesh contains the adjacency
matrix of the transitive closure after $n$ cycles. Note
that this algorithm does not use the mesh connections but
only the buses.

Several algorithms exist to compute transitive closure
on a plain mesh, ([1], [5], [8]), but as far as simplicity
is concerned, none of them come close to the algorithm
above. It is true, that after we transform our algorithm
to run on a plain mesh it will become complex as well.
But this translation can be done by a machine!

Let us see how to eliminate the busses from a given
bussing mesh $Y$ executing a normal algorithm. As in a
plain mesh, each node has a self-loop, and every edge
has a single register. By adding dummy edges, make
$Y$ a torus and homogenize it, generating $Y'$. Since $Y'$
is homogeneous, we may assume the initial positions of
the master-row and master-column to be the first row
and column. By rewiring and an additional trick we will
transform $Y'$ into $Y''$ such that in $Y''$ the master-row
and master-column are stationary, and are the first row
and column.

Rewire $Y'$ with the automorphism that maps $<i,j>$
to $<i-1,j-1>$, generating $Y''_d$. The “$d$” stands for the
<down, right> direction. (We need to extend the concept
of automorphism to take care of busses.) If the algorithm
moves the mastership constantly in the <down, right> direc­
tion, then in $Y''_d$ the mastership is static. Similarly,
for any possible direction of move there is an automor­
phism that cancels this move, and a system $Y''_r$ rewired
illustrate that it is based on rewiring.

ulated on a de Bruijn graph of the same size and at the following holds.

up with a systolic torus which has diagonal connections with a slowdown of two [8].

Each combinational unit 16 times larger than the original.

More about that in the full paper. In both cases, we end up with a systolic system that is otherwise systolic: one of Leiserson and Saxe [9], the other of Even and Litman [3]. In the case of $Y''$, but not in general, these techniques can be extended to eliminate all the busses, making the system a systolic one. The technique of [9] slows the system by a factor of three. The technique of [3] does not slow the system. It reduce each dimension of the torus by four, and make each combinational unit 16 times larger than the original. Moreover, in both cases, we end up with a systolic torus which has diagonal connections as well. Such a torus can be converted to a plain mesh with a slowdown of two [8].

9 The Hypercube and De Bruijn graph

An algorithm of the hypercube is normal [8] if the following holds.

- In any cycle, the algorithm uses only edges of a single dimension, called the current dimension.
- After each cycle, the current dimension moves one position either up or down (with wraparound).
- All nodes, even idle ones, "know" which direction the current dimension moves.
- The initial position of the current dimension is independent of the data.

Let $Y$ be an n-dimensional hypercube. As usual, consider its nodes to be binary words of length $n$. For a node $\omega$, let $\pi(\omega)$ be the result of shifting $\omega$ one bit to the right in a circular fashion. Homogenize the hypercube. Rewire it with the automorphisms $\pi$ and $\pi^{-1}$ combined (as in the previous section), and call the resulting system $Y'$. The edges of $Y'$ originate either from self-loops or inter-node edges of $Y$. Most of the edges of $Y'$ of the latter type are never used — except of a single dimension of $Y$, all the edges of $Y'$ that originate from other dimension always idle. Discard these edges. The result is a systolic de Bruijn graph.

10 Acknowledgements

I wish to thank Shimon Even and Assaf Schuster for helpful discussions. Special thanks goes to Guy Even for starting it all.

References


