Chapter 1

PARTITIONING AND PLACEMENT FOR BUILDABLE QCA CIRCUITS

Sung Kyu Lim
School of Electrical and Computer Engineering
Georgia Institute of Technology
limsk@ece.gatech.edu

Mike Niemier
College of Computing
Georgia Institute of Technology
mniemier@cc.gatech.edu

Abstract Quantum-dot Cellular Automata (QCA) is a novel computing mechanism that can represent binary information based on spatial distribution of electron charge configuration in chemical molecules. It has the potential to allow for circuits and systems with functional densities that are better than end of the roadmap CMOS, but also imposes new constraints on system designers. In this article, we present the first partitioning and placement algorithm for automatic QCA layout. The purpose of zone partitioning is to initially partition a given circuit such that a single clock potential modulates the inter-dot barriers in all of the QCA cells within each zone. We then place these zones as well as individual QCA cells in these zones during our placement step. We identify several objectives and constraints that will enhance the buildability of QCA circuits and use them in our optimization process. The results are intended to: (1) define what is computationally interesting and could actually be built within a set of predefined constraints, (2) project what designs will be possible as additional constructs become realizable, and (3) provide a vehicle that we can use to compare QCA systems to silicon-based systems.

Keywords: Quantum-dot Cellular Automata, partitioning, placement

Introduction

Nano technology and devices will have revolutionary impact on the CAD field. Similarly, computer-aided design (CAD) research at circuit, logic and architectural levels for nano devices can provide valuable feedbacks to nano research and illuminate ways for developing new nano devices. It is time for CAD researchers to play an active role in nano research. One approach to computing at the nano-scale is the quantum-dot cellular automata (QCA) concept that represents information in a binary fashion, but replaces a current switch with a cell having a bi-stable charge configuration. QCA devices can be realized in metal [Amlani et al., 1998], or with chemical molecules [Lieberman et al., 2002]. A wealth of experiments have been conducted with metal-dot QCA, with individual devices [Amlani et al., 1998], logic gates [Snider et al., 1999] [Amlani et al., 1999] wires [Snider et al., 1999], latches [Kummamuru et al., 2002], and clocked devices [Kummamuru et al., 2002], all having been realized. This advancement is followed by various recent efforts in developing CAD tools for QCA based circuits and systems [Gergel et al., 2003] [Bernstein, 2003].

Our goal in this article is to explain how CAD can help research to move from small circuits to small systems of quantum-dot cellular automata (QCA) devices. We leverage our ties to physical scientists who are working to build real QCA devices. Based upon this interaction, a set of near-term *buildability constraints* has evolved - essentially a list of logical constructs that are viewed as implementable by physical scientists in the nearer-term. Until recently, most of the design optimizations have been done by hand. Then these initial attempts to automate the process of removing a single, undesirable, and unimplementable feature from a design were quite successful. We now intend to use CAD, especially physical layout automation, to address all undesirable features of design that could hinder movement toward a "buildability point" in QCA. The net result should be an expanded subset of computationally interesting tasks that can be accomplished within the constraints of a given buildability point. CAD will also be used to project what is possible as the state-of-the-art in physical science expands.

In this article, we present the first partitioning and placement algorithm for automatic QCA layout. The purpose of zone partitioning is to initially partition a given circuit such that a single clock potential modulates the inter-dot barriers in all of the QCA cells within each zone. We then place these zones as well as individual QCA cells in these zones during our placement step. We identify several objectives and constraints that will enhance the buildability of QCA circuits and use them in our

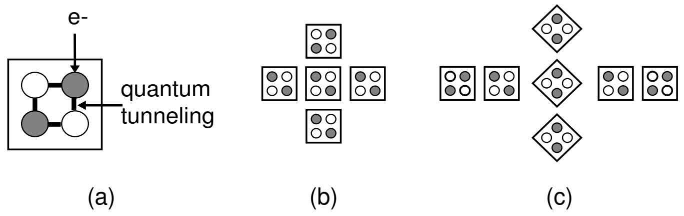


Figure 1.1. Illustration of QCA device, majority gate, and wires.

optimization process. The results are intended to: (1) define what is computationally interesting and could actually be built within a set of predefined constraints, (2) project what designs will be possible as additional constructs become realizable, and (3) provide a vehicle that we can use to compare QCA systems to silicon-based systems.

1. Preliminaries

This section briefly reviews the background of QCA devices and clocking scheme. Then, a detailed comparison between QCA and CMOS technologies is provided. Lastly, we discuss the need for QCA CAD research, especially the physical layout automation.

QCA Circuit Building Blocks

QCA circuits are built from the following components.

QCA Device. A high-level diagram of a "candidate" four-dot metal QCA cell appears in Figure 1.1(a) [Amlani et al., 1998]. It depicts four quantum dots that are positioned to form a square. Exactly two mobile electrons are loaded into this cell and can move to different quantum dots by means of electron tunneling. Coulombic repulsion will cause "classical" models of the electrons to occupy only the corners of the QCA cell, resulting in two specific polarizations. These polarizations are configurations where electrons are as far apart from one another as possible, in an energetically minimal position, without escaping the confines of the cell.

QCA Logic Gate. QCA's logic functionality will be explained in terms of "generic" 4-dot cells. The fundamental QCA logical gate is the three-input majority gate. It consists of five cells and implements the logical equation AB+BC+AC as shown in Figure 1.1(b). Computation is performed by driving the device cell to its lowest energy state, which will occur when it assumes the polarization of the majority of the three input cells. Here, the electrostatic repulsion between the electrons in the three input cells, and the electrons in the device cell will be at a minimum. As the majority function can be reduced to the AND and OR function, and a means for signal inversion is possible, QCA's logic set is functionally complete.

QCA Wire. One way of moving data from point A to point B in a QCA circuit is with a 90-degree wire. The wire is called "90-degrees" as the cells from which it is made up are oriented at a right angle. The wire is a horizontal row of QCA cells and a binary signal propagates from left-to-right because of electrostatic interactions between adjacent cells. A QCA wire can also be comprised of cells rotated 45-degrees. Here, as a binary signal propagates down the length of the wire, it alternates between a binary 1 and a binary 0 polarization. QCA wires possess the unique property that they are able to cross in the plane without the destruction of the value being transmitted on either wire as shown in Figure 1.1(c). This property holds only if the QCA wires are of different orientations (i.e. a 45-degree wire crossing a 90-degree wire). However, it is most important as at present, all layout is assumed to be two-dimensional.

QCA Clock. QCA's clock was first characterized by Lent, et. al. as having 4 phases. During the first clock phase (switch), QCA cells begin un-polarized with inter-dot potential barriers low. During this phase barriers are raised, and the QCA cells become polarized according to the state of their drivers (i.e. their input cells). It is in this clock phase, that actual switching (or computation) occurs. By the end of this clock phase, barriers are high enough to suppress any electron tunneling and cell states are fixed. During the second clock phase (hold), barriers are held high so the outputs of the subarray that has just switched can be used as inputs to the next stage. In the third clock phase, (release), barriers are lowered and cells are allowed to relax to an unpolarized state. Finally, during the fourth clock phase (relax), cell barriers remain lowered and cells remain in an unpolarized state [Tougaw and Lent, 1994].

Individual QCA cells need not be clocked or timed separately. However, a physical array of QCA cells can be divided into zones that offer the advantage of multi-phase clocking and group pipelining. For each zone, a single potential would modulate the inter-dot barriers in all of the cells in a given zone. Such a clocking scheme allows one zone of QCA cells to perform a certain calculation, have its state frozen by the raising of inter-dot barriers, and then have the output of that zone act as the input to a successor zone.

QCA Wins

As QCA is being considered as an alternative to silicon-based computation, it is appropriate to enumerate what QCA's "wins" over silicon-based systems could be (as well as its potential obstacles). We begin by listing obstacles to CMOS-based Moore's Law design (Table 1.1), their effects on silicon based systems, and how they will affect QCA.

Based on the information in Table 1.1 it is apparent that QCA faces some of the same general problems as silicon-based systems (timing issues, lithography resolutions, and testing), that QCA does not experience some of the same problems as silicon-based systems (quantum effects and tunneling), and that silicon-based systems can address one problem better than QCA currently can (I/O) However, if the I/O problem is resolved, QCA can potentially offer significant "wins" with regard to reduced power dissipation and fabrication. Additionally, QCA can also offer orders of magnitude in potential density gains when compared to silicon-based systems. When examining the existing design of an ALU for a simple processor [Niemier and Kogge, 2001], one version is potentially 1800 times more dense (assuming deterministic cell placement) than an end of the CMOS curve equivalent (0.022 micron process). If based on a more implementable FPGA (whose logic cell is a single NAND gate), the ALU is no less dense than a fully custom, end of the CMOS curve equivalent [Niemier and Kogge, 2004]. Clearly, realizable and potential QCA systems warrant further study.

Builability Analysis via QCA CAD

One might argue that it would be premature to perform any systems-level study of an emergent device while the physical characteristics of a device continue to evolve. However, it is important to note that many emergent, nano-scale devices are targeted for computational systems - and to date, most system-level studies have been proposed by physical scientists, and usually end with a demonstration of a functionally complete logic set or a simple adder. Useful and efficient computation will involve much more than this, and, in general, it is important to provide scientists with a better idea of how their devices should function. This coupling can only lead to an accelerated development of functional and interesting systems at the nano-scale. More specifically, with QCA, physicists are currently preparing to test the self-assembly process and its building blocks. Thus, our work can help provide the physicists with

Table 1.1. Comparing characteristics of silicon-based systems to QCA based systems

Obstacle	Effect on CMOS circuits	How it relates to QCA
Quantum Effects and Tunneling	A gate that controls the flow of electrons in a transistor could allow them to tunnel through small barriers - even if the device is supposed to be off [Packan, 1999].	No effect; QCA devices are charge containers, not current switches and actually leverage this property.
High power dissipation	Chips could melt [Rabaey, 1996] [Mead and Conway, 1980] unless problems are overcome for which SIA roadmap says "there are no known solutions". 2014 projection: chip with 10^{10} devices dissipates 186W of power.	10^{11} QCA devices with 10^{-12} switching times dissipate 100W of power. QCA's silicon-based clock will also dissipate power. Still, clocking wires should move charge adiabatically [Hennessy and Lent, 2001], greatly reducing power consumption.
Slow wires	Wires continue to dominate the overall delay [Ho et al., 2001]. Also, projections show that for 60 nm feature sizes, less than 10% of the chip is reachable in 1 clock cycle [Hamilton, 1999].	The inherent pipelining caused by the clock make global communication and signal broadcast difficult [Niemier, 2003]. Problems are similar to silicon-based systems but for different reasons.
Lithography resolutions	Shorter wavelengths and larger apertures are needed to provide finer resolutions for decreased feature sizes.	QCA's clock wiring is done lithographically, which is subject to the same constraints as silicon-based systems. However, closely spaced nanowires could also be used [Lieberman et al., 2002].
Chip I/O	I/O count continue to increases as the technology advances (Rent's rule), but pin counts do not scale well. With more processing power, we will need more I/O [Rabaey, 1996].	I/O remains under investigation with one approach being to include "sticky ends" at the ends of certain DNA tiles in order to bind nano-particles or nanowires
Testing	Even if designs are verified and simulated, defects caused by impurities in the manufacturing process, misalignment, broken interconnections, etc. can all contribute to non-functional chips. Testing does not scale well [Rabaey, 1996].	We must find and route around defects caused by self-assembly and/or find new design methodologies to make circuits robust. Defects for self-assembled systems could range from 70% to 95%. Structures such as thicker wires could help.
Cost	Fabrication facility cost doubles approximately every 4.5 years [Rutten, 2001], and could reach 200 billion dollars in 2015.	Self-assembly could be much more inexpensive.

computationally interesting patterns - the real and eventual desired end result.

Our toolset will focus on the following undesirable design schematic characteristics associated with a near-to-midterm buildability point: large amounts of deterministic device placement, long wires, clock skew, and wire crossings. We will use CAD to: (1) identify logic gates and blocks that can be duplicated to reduce wire crossings, (2) rearrange logic gates and nodes to reduce wire crossings, (3) create shorter routing paths to logical gates (to reduce the risk of clock skew and susceptibility to defects and errors), and (4) reduce the area of a circuit (making it easier to physically build). Some of these problems have been individually considered in existing work for silicon-based VLSI design, but in combination, form a set of constraints unique to QCA requiring a unique toolset to solve them.

CMOS vs QCA Placement

Although QCA and CMOS have considerable technological differences, CMOS VLSI placement algorithms have been modified to satisfy the design constraints imposed by QCA physical science. There are many reasons for using this approach. Notably, VLSI design automation algorithms work on graph-based circuits, and it has been found to be advantageous to represent QCA circuits as graphs–especially given that at present, only two dimensional circuits have been proposed and are seen as technically feasible. Existing algorithm can be fine-tuned to meet QCAs constraints and objectives. Additionally, physical design issues for CMOS have been widely studied, optimized, and proven to be NP-complete. Thus, it makes sense to leverage this existing body of knowledge and apply it to a new problem. Finally, because so few design automation tools and methodologies exist for QCA, using VLSI algorithms as a base will allow us to compare and set standards for our place and route methodologies.

More specifically, we note the following similarities and differences between CMOS and QCA placement.

- Similarity: In CMOS placement, partitioning, floorplanning, and placement are performed in this order (so called hierarchical approach) to efficiently handle the design complexity. We use a similar approach in QCA placement: zone partitioning, zone placement, and cell placement. The following objectives are common in both CMOS and QCA partitioning for the same purpose: cutsize and performance. The area, performance, congestion, and wirelength objectives are common in both CMOS and QCA placement.

- Difference: two major sources of the difference are QCA clocking and QCA single-layer routing resource. Wire crossing minimization is critical in QCA placement since QCA layout needs to be done in a single layer unlike the multi-layer CMOS layout. Thus, node duplication in CMOS is targeting area and performance while QCA duplication focuses on wire crossing. In order to meet the QCA clocking requirement, we use *k-layered bipartite graphs* to represent the original and partitioned netlist. This in turn requires QCA partitioning to minimize area increase (after the bipartite graph construction). In addition, the length of all reconvergent paths from the same partition should be balanced (to be discussed in detail later) and cyclic dependency is not allowed.

2. Problem Formulation

In this section, we provide an overview of the placement process of QCA circuits. We then present the formulation of three problems related to QCA placement–zone partitioning, zone placement, and cell placement problem. Our recent work on zone partitioning and zone placement work is available in [Nguyen et al., 2003] and cell placement in [Ravichandran et al., 2004].

Overview of the Approach

QCA placement is divided into three steps: zone partitioning, zone placement, and cell placement. The purpose of zone partitioning is to decompose an input circuit such that a single potential modulates the inner-dot barriers in all of the QCA cells that are grouped within a clocking zone. Unless QCA cells are grouped into zones to provide zone-level clock signals, each individual QCA cell will need to be clocked. The wiring required to clock each cell individually would easily overwhelm the simplicity won by the inherent local interconnectivity of QCA architecture. However, because the delay of the biggest partition also determines the overall clock period, the size of each partition must also be determined carefully. In addition, four-phase clocking imposes a strict constraint on how to perform partitioning. The zone placement step takes as input a set of zones–with each zone assigned a clocking label obtained from zone partitioning. The output of zone placement is the best possible layout for arranging the zones on a two dimensional chip area. Finally, cell placement visits each zone to determine the location

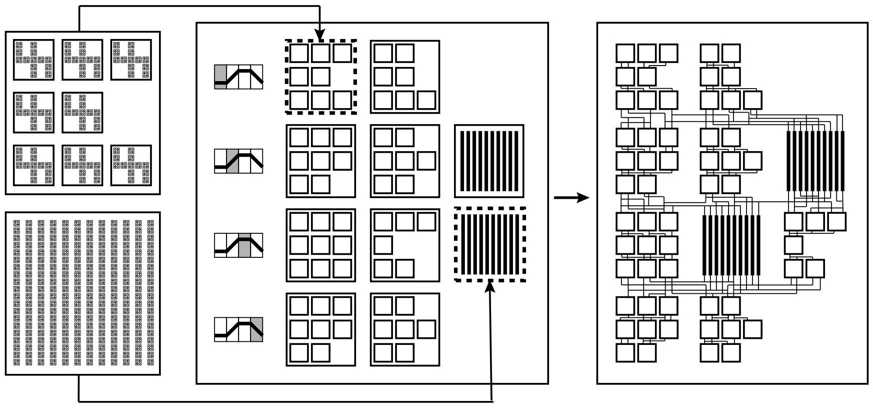


Figure 1.2. Overview of the QCA layout automation process. A logic and a wire block are shown. First, the input circuit is partitioned into logic and wire blocks (zone partitioning). Second, each block is placed onto 2D space while satisfying QCA timing constraints (zone placement). Third, QCA cells in each block is placed (QCA cell placement). Fourth, routing is performed to finish inter-block interconnect (global QCA routing) and intra-block interconnect (detailed QCA routing).

of each individual logic QCA cell–a cell used to build majority gates. An illustration of QCA placement and routing step is shown in Figure 1.2.

Zone Partitioning Problem

A gate-level circuit is represented with a directed acyclic graph (DAG) $G(V, E)$. Let P denote a partitioning of V into K non-overlapping and non-empty blocks. Let $G'(V', E')$ be a graph derived from P, where V' is a set of logic blocks and E' is a set of cut edges based on P. A directed edge $e(x, y)$ is called *cut* if x and y belong to different blocks in P. Two paths p and q in G' are *reconvergent* if they diverge from and reconverge to the same blocks. If $l(p)$denotes the length of a reconvergent path p in G', then $l(p)$ is defined to be the number of cut edges along p.

DEFINITION 1.1 *Zone partitioning: we seek a partitioning of logic gates in the given netlist into a set of zones so that cutsize (= total number of cut nets), wire block (= required during the subsequent zone placement) are minimized. The area of each partition needs to be bounded (area constraint), and there should not exist cyclic dependency among partitions (acyclic constraint). In addition, the length of all reconvergent paths should be balanced (clocking constraint).*

An illustration of reconvergent path constraint is shown in Figure 1.3. Cycles may exist among partitions as long as their lengths are in

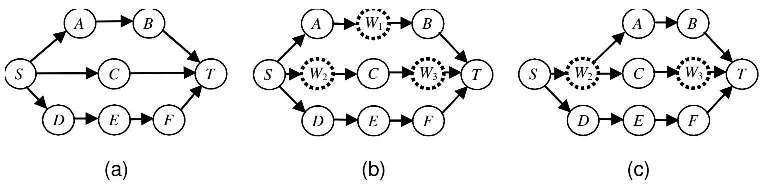


Figure 1.3. Illustration of reconverent path constraint. (a) all three reconvergent paths from S T are unbalanced. If S is in the switch phase, A, B, and T will be in relax, release, and hold phase. This puts C and T into relax and release, thereby causing a conflict at T. The bottom path forces T to be in switch phase, causing more conflict. (b) wire blocks W_1, W_2, and W_3 are inserted to resolve this QCA clocking inconsistency. (c) some wire blocks are shared to minimize the area overhead.

multiples of four due to QCA clocking. However, it is hard to enforce this constraint while handling other objectives and constraints. Therefore, we decide to prevent any cycles from forming at the partition level. In addition, it is difficult to maintain the reconvergent path constraint during partitioning process. Therefore, we allow the reconvergent path constraint to be violated and perform a post-process to add *wire blocks* to fix this problem. Since the addition of wire blocks cause the overall area to increase, we minimize the amount of wire blocks that are needed to completely remove the reconvergent path problems during zone partitioning.

Zone Placement Problem

Assuming that all partitions (= zone) have the same area, placement of zones becomes a geometric embedding of the partitioned network onto a $m \times n$ grid, where each logic/wire block is assigned to a unique location in the grid. The resulting partitioned network after zone partitioning and wire block insertion satisfies the following condition: $lev(x) = lev(y) + 1$ for each inter-partition edge $e = (x, y)$. In this case, a bipartite graph exists for every pair of neighboring clocking levels. Therefore, our grid placement problem is to embed this *zone-level k-layered bipartite graph* onto an $m \times n$ grid so that all blocks in the same layer are placed in the same row. All the I/O terminals are assumed to be located on the top and bottom boundary of each block, and we may insert routing channels between clocking levels for the subsequent routing.

DEFINITION 1.2 *Zone placement: we seek a placement of zones we obtain from zone partition onto a 2D space so that area, wire crossing and wirelength are minimized. Each zone (= logic/wire block) is labeled with QCA clocking level (= longest path length from input zones), and all* *zones with the same QCA clocking level should be placed in the same row (clocking constraint). In addition, all inter-zone wires need to connect two neighboring rows (neighboring constraint).*

Cell Placement Problem

The input to the cell placement is zone placement result, where all logic/wire blocks at the same clocking level are placed in the same row. Then the output of cell placement is an arrangement of QCA cells in each logic block. The reconvergent path problem does not exist in cell placement–it is perfectly fine to have unbalanced reconvergent path lengths among the logic gates in each logic block. The reason is that correct output values will eventually be available at the output terminals in each block if the clock period is longer than the maximum path delay in each block. We determine the clock period based on the maximum path delay among all logic/wire blocks.

DEFINITION 1.3 *Cell placement: we seek a placement of individual logic gates in the logic block so that area, wire crossing and wirelength are minimized. The following set of constraints exists during QCA cell placement: (1) the timing constraint: the signal propagation delay from the beginning of a zone to the end of a zone should be less than a clock period established from zone partitioning and the constraints of physical science (maximum zone delay) (i.e. we want to eliminate possible skew), (2) the terminal constraint: the I/O terminals are located on the top and bottom boundaries of each logic block, (3) the signal direction constraint: the signal flow among the logic QCA cells needs to be unidirectional-from the input to the output boundary for each zone.*

The signal direction is caused by QCA's clocking scheme, where an electric field E created by underlying CMOS wire is propagating in unidirectionally within each block. Thus, cell placement needs to be done in such a way to propagate the logic outputs in the same direction as E. In order to balance the length of intra-zone wires, we construct *cell-level k-layered bipartite graph* for each zone and place this graph.

3. Zone Partitioning Algorithm

This section presents our zone partitioning and wire block insertion algorithms. Our zone partitioning algorithm is iterative improvement based method, whereas our wire block insertion is based on longest path computation.

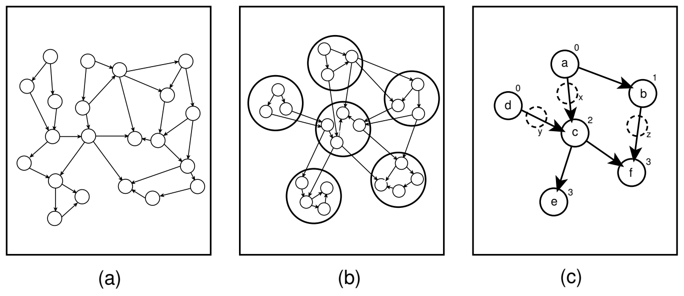


Figure 1.4. Illustration of zone partitioning and wire block insertion. (a) directed graph model of input circuit, (b) zone partitioning under acyclicity and reconvergent path constraint, (c) wire block insertion, where the numbers denote the longest path length. The dotted nodes indicate wire blocks.

Zone Partitioning

Let $lev(p)$ denote the longest path length from the input partitions (partitions with no incoming edges) to partition p, where the path length is the number of partitions along the path. Then $wire(e)$ denotes the total number of wire blocks to be inserted on an inter-partition edge e to resolve the unbalanced reconvergent path problem (clocking constraint of the QCA zone partitioning problem). Simply, $wire(e) = lev(y) + lev(x) - 1$ for $e = (x, y)$, and the total number of wiring blocks required without resource sharing is $\sum wire(e)$. Thus, our heuristic approach is to minimize the $\sum wire(e)$ among all inter-zone edges while maintaining acyclicity. Then, during post-processing, any remaining clocking problems are fixed by inserting and sharing wire blocks. An illustration of zone partitioning and wire block insertion is shown in Figure 1.4.

First, the cells are topologically sorted and evenly divided into a number of partitions $(p_1, p_2, \cdots p_k)$. The partitions are then level numbered using a breadth-first search. Next, the acyclic FM partitioning algorithm [Cong and Lim, 2000] is performed on adjacent partitions p_i and p_{i+1}. Constraints that must be met during any cell move include area and acyclicity. The cell gain has two components: cutsize gain and wire block gain. The former indicates the reduction in the number of inter-partition wires, whereas the latter indicates the reduction in the total number of wire blocks required. We then find the best partition based on a combined cost function for both cutsize and wire block gain. Multiple passes are performed on two partitions p_i and p_{i+1} until there is

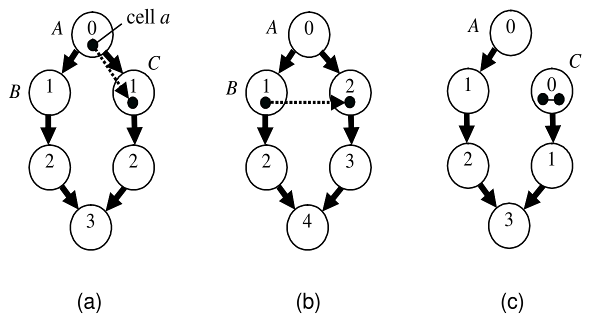


Figure 1.5. Illustration of clocking level update.

no more improvement on the cost. Then, this acyclic bipartitioning is performed on partitions p_{i+1} and p_{i+2}, etc.

Movement of a single cell can possibly change $lev(p)$, the level number of a partition p. Therefore every time a cell move is made, we check to see if this cell move affects the level number. There are two ways levels can change: an inter-zone edge is newly introduced or completely removed. In Figure 1.5, cell a in Figure 1.5(a) is moved from partition A to B, thereby creating a new inter-partition edge in 1.5(b). This in turn changes the level of all downstream partitions. In Figure 1.5(c), cell a in Figure 1.5(a) is moved from partition A to C, thereby removing the inter-partition edge between A and C 1.5(c). This again changes the level of all downstream partitions. For updating the level, we maintain a maxparent for each p so that the level number of the parent of p is $lev(p) - 1$. $lev(F)$ is defined as the level number of the "from block" of a cell c and $lev(T)$ is defined as the level number of the "to block" of c. In the first case where a new inter-partition edge is created, $lev(T)$ is updated if $lev(F) \geq lev(T)$ after the cell move. In this case, $lev(T) = lev(F) + 1$. Then, we recursively update the maxparent and levels of all downstream partitions. The maxparent for partition C was changed from A to B in Figure 1.5(b), and $lev(C)$ now becomes $lev(B) + 1 = 2$. This in turn requires the level number of all downstream nodes to change. In the second case where an existing inter-partition edge is removed, the maxparent again needs to be update. The maxparent for partition C was changed from A to none in Figure 1.5(c), and $lev(C)$ now becomes $lev(C) = 0$.

Wire Block Insertion

During the post-processing, we fix any remaining clocking problems by inserting and sharing wire blocks, while satisfying wire capacity constraints. The input to this algorithm is the set of partitions and inter-partition edges. First, a super-source node is inserted in the graph whose fan-out neighbors are the original sources in the graph. This is done to ensure that all sources are in the same clocking zone. Then the single-source longest path is computed for the graph with the super-source node as the source–and every partition is assigned a clocking level based on its position in the longest path from the source. For a graph with E' inter-partition edges, this algorithm runs in exactly $O(E')$ iterations. In the algorithm's next stage, any edge connecting partitions that are separated by more than one clock phase is marked, and the edge is added to an array of bins at every index where a clocking level is missing in the edge. The following algorithms perform wire block insertion.

```
**wire_block_insertion**($G(V, E)$)
    $lev(SUPER) = -1$;
    $Q$.enque($SUPER$);
    BFS-mark($G, SUPER$);
    while ($E$ not empty)
        $N = E$.pop();
        $S = lev(N.source)$;
        $T = lev(N.sink)$;
        while ($S + 1 < T$)
            $S = S + 1$;
            $BIN[S] = (BIN[S], E)$;

**BFS-mark**($G, Q$)
    $N = Q$.deque;
    $S$ = set of fanout neighbors of $N$;
    while ($S$ not empty)
        $A = S$.pop();
        if (LAST-PARENT($A$) = $N$)
            $lev(A) = lev(N) + 1$;
            $Q$.enque($A$);
    BFS-mark($G, Q$)
```

The number of wire blocks in each bin is calculated based on a predetermined capacity for the wire blocks. This capacity is calculated based on the width of each cell in the grid. Then the inter-partition edges are distributed amongst the wire block, filling one wire block to full capacity before filling the next. It might seem that a better solution would be to evenly distribute the edges to all the wire blocks in the current level. This is not true because the wire blocks with the most number of feed-throughs are placed closer to the logical blocks in the next stage. This minimizes wirelength, and hence the number of wire crossings.

4. Zone Placement Algorithm

This section presents our zone placement algorithm. Our zone partitioning algorithm is iterative improvement based method, where the initial placement of zone-level k-level-bipartite-graph is refined via block swap for wire crossing and wirelength reduction.

Placement of k-Layered Bipartite Graph

The logical blocks (obtained from the partitioning stage) and the wire blocks (obtained from post-processing) are placed on an $m \times n$ grid with a given aspect ratio and skew. The individual zone dimensions and the column widths are kept constant to ensure scalability and manufacturability of this design as clocking lines would have to be laid underneath QCA circuits with great precision. The partitions are laid out on the grid, with the cells belonging to the first clocking zone occupying the leftmost cells of the first row of the grid, and the next level occupying the leftmost cell of the next row, etc., until row r. The next level of cells is placed again on row r to the right of the rightmost placed cell amongst the r placed rows. The next level of cells is placed in row $r-1$ and the rest of the cells are placed in a similar fashion until the first row is reached. This process is repeated until all cells are placed (thereby forming a "snake-shape"). The white nodes are white space that is introduced because of variations in the number of wire and logic blocks among the various clocking levels. The maximum wirelength between any two partitions in the grid determines the clock frequency for the entire grid as all partitions are clocked separately. For the first and last rows (where inter-partition edges are between partitions in two different columns), maximum wirelength was given more priority as maximum wire length at these end zones can be twice as bad as the maximum wire length between partitions on the same column. An illustration of zone placement and wire crossing minimization is shown in Figure 1.6.

Wire Crossing Minimization

During the next phase, blocks are reordered within each clocking level to minimize inter-partition wirelength and wire crossings. Two classes of solutions were applied to minimize the above objectives: an analytical solution that uses a weighted barycenter method, and Simulated

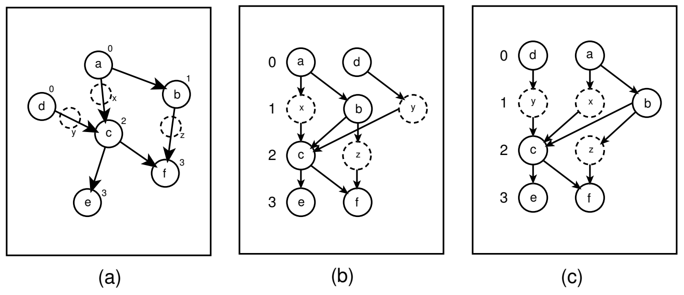


Figure 1.6. Illustration of zone placement and wire crossing minimization. (a) zone partitioning with wire block insertion, (b) zone placement, where a zone-level k-layered bipartite graph is embedded onto a 2D space, (c) wire crossing minimization via block re-ordering.

Annealing. The analytical method only considers wire crossings since as there is a strong correlation between wirelength and number of wire crossings.

Analytical Solution: A widely used method for minimizing wire crossings (introduced by Sugiyama et al. [Sugiyama et al., 1981]) is to map the graph into k-layer bipartite graph. The vertices within a layer are then permuted to minimize wire crossings. This method maps well to this problem as we need to only consider the latter part of the problem (the clocking constraint yields us the k-layer bipartite graph). Still, even in a two-layer graph, minimizing wire-crossings is NP-hard [Sugiyama et al., 1981]. Amongst many heuristics proposed, the *barycenter heuristic* [Sugiyama et al., 1981] has been found to be the best heuristic in the general case for this class of problems. A modified version of the barycenter heuristic was used to accommodate for edge weights. The edge weights represent the number of inter-partition edges that exist between the same pair of partitions. The heuristic can be summarized as follows:

$$barycenter(v) = \frac{\sum_N [weight(n) \times position(n)]}{\sum_N weight(n)}$$

where v is the vertex in the variable layer, n is the neighbor in the fixed layer, and N is the set of all neighbors in the fixed layer.

Simulated Annealing: A move is done by randomly choosing a level in the graph and then swapping two randomly chosen partitions $[p_1, p_2]$ in that level in order to minimize the total wirelength and wire crossing. In our implementation, the initial calculation of the wirelength takes $O(n)$ and updating wire crossing takes $O(n^3)$ where n is the number of nodes in a layer of the bipartite graph. In our approach, we initially compute

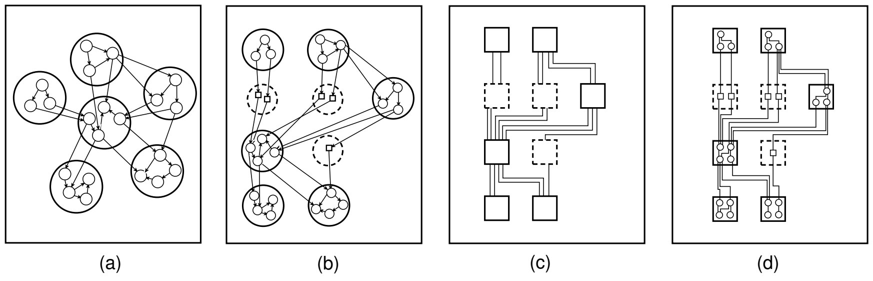


Figure 1.7. Illustration of cell placement, global routing and detailed routing. (a) zone placement result, (b) cell placement result, where cells in each partition (= zone) again forms a k-layered bipartite-graph, (c) global routing, where inter-zone connections are made, (d) detailed routing, where intra-zone connections are made.

the wirelength and wire crossing and incrementally update these values after each move so that the update can be done in $O(m)$ time where m is the number of neighbors for p_i. This speedup allows us to explore a greater number of candidate solutions, and as a result, obtain better quality solutions.

5. Cell Placement Algorithm

This section presents our cell placement algorithm, which consists of feed-through insertion, row folding, and wire crossing and wirelength optimization steps. Figure 1.7 shows an illustration of cell placement as well as QCA routing.

Feed-Through Insertion

In order to satisfy the relative ordering and to satisfy the signal direction constraint, the original graph $G(V, E)$ is mapped into a k-layered bipartite graph $G'(V', E')$ which is obtained by insertion of feed-through gates, where V' is the union of the original vertex set V and the set of feed-through gates, and E' is the corresponding edge set. The following algorithm performs feed-through insertion.

```
feed-through_insertion($G(V, E)$)
    if ($V$ is empty)
        return;
    $n = V$.pop();
    if ($n$ has no child with bigger level)
        return;
```

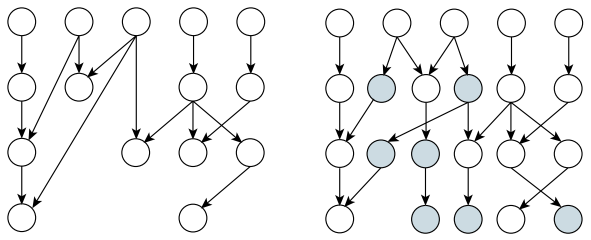


Figure 1.8. Illustration of feed-through insertion, where a cell-level k-layered bipartite-graph is formed via feed-through nodes.

```
$g$ = new feed-through;
$lev(g) = lev(n) + 1$;
for (each child $c$ of $n$)
    $g$ = parent($c$);
    $c$ = child($g$);
$n$ = parent($g$);
$g$ = child($n$);
add $g$ into $G$;
feed-through_insertion($G(V, E)$)
```

In this algorithm, we traverse through every vertex in the vertex set of the graph. For a given vertex, if any of the outgoing edges terminate at a vertex with topological order more than one level apart, a new feed-through vertex is added to the vertex set. The parent of the feed-through is set to the current vertex, and all children of the current vertex which have a topological order difference of more than one is set as the children of the feed-through. We do not need to specifically worry about the exact level difference between the feed-through and the child nodes, since this feed-through insertion is a recursive process. This algorithm runs in $O(k|V'|)$, where k is the maximum degree of V'. Figure 1.8 shows the graph before and after feed-through insertion. A trivial result of this stage is that all short paths have a set of feed-throughs between the last logical gate in the path and last row.

Row-folding Algorithm

After the feed-through insertion stage, some rows may have more gates than the average number of gates per row. The row with the largest number of gates defines the width of the entire zone, and hence the width of the global column that the zone belongs to. This increases the circuit area by a huge factor. Hence, rows with a large number of cells are folded into two or more rows. This is done by inserting feed-through gates in place of the logic gates and moving the gates to the next row. Row-folding decreases the width of the row since feed-throughs have a lower width than the gate it replaces. A gate g is moved into the next existing row if it belongs to the row that needs to be folded and all paths that g belongs to contain at least one feed-through with a higher topological order than g. The reason for the feed-through condition is that g, along with all gates between g and the feed-through can be pushed to a higher row, and the feed-through can be deleted without violating the topological ordering constraint. The following algorithm performs row folding.

```
row_folding($G, w$)
    if ($w$ is a feed-through)
        return(TRUE);
    if ($w$.level = $G$.max_level)
        return(FALSE);
    RETVAL = TRUE;
    $k = w$.out-degree;
    $i = 0$;
    while (RETVAL and $i < k$)
        RETVAL = row_folding(G,$w$.CHILD(i));
        $i = i + 1$;
    return(RETVAL);
```

This algorithm returns true if a node can be moved, and false if a new row has to be inserted. If this feed-through criterion is not met, and the row containing g has to be folded, then a new row is inserted and g is moved into that row.

The number of gates that need to be moved from a row that needs folding to a new row is given by the following trivial calculation. Let n be the number of gates that need to be moved to the next row. Let m be the original number of gates in the row, and let M be the maximum number of gates allowed in a row. Further, let a be the ratio of the width of a feed-through to the width of the gate. Since width of a gate is always greater than the width of a feed-through, $a < 1$. For every gate that is moved to a new row, a feed-through has to be inserted in its original place. Hence, after moving n gates to the next row, the width of the original row will now be $m - n + an$, so $n = (m - M)/(1 - a)$. This calculation is repeated for the next row if n is itself greater than the constraint M. The principal reason for increasing the height of a zone rather than increasing the width of the zone is that the width of global column that the zone belongs to is much smaller than height of the column since the aspect ratio of the entire circuit layout is close to unity.

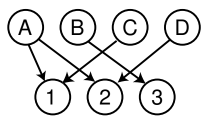
wirelength and Wire Crossing Minimization

During zone placement stage, a zone-level k-layered bipartite graph is formed via wire block insertion. This graph is then placed in such a way that all zones at the same clocking level is placed in the same row. The same graph transformation and placement is done during cell placement–a cell-level k-layered bipartite graph is formed via feed-through insertion, and this graph is placed in such a way that all cells of the same longest path lenegh are placed in the same row. In both cases, iterative improvement is performed to reduce the wire crossing and wirelength at the zone and cell level. We perform barycenter heuristic to build the initial solution and perform block/cell swaps to improve the solution quality.

To compute the net wirelength in a circuit we traverse through every vertex and accumulate the difference between the column numbers of the vertex and all of its children. This runs in $O(N)$, where N is the number of vertices. But, during the first calculation, we store the sum of all outgoing wirelength in every vertex. This enables us to incrementally update if the position of only one node changes. A node cannot change its row number since at this stage the topological level is fixed. If a node changes its position within a level, then it is enough to calculate the difference in position with respect to its neighbors alone. Hence, subsequent wirelength calculation is reduced to $O(K)$ where K is the node's vertex degree.

Wire crossing computation can be done with either the adjacency list or matrix, depending on the sparseness of the graph. We used the adjacency matrix to compute the number of wire crossings in a graph. In a graph, there is a wire crossing between two layers v and u if v_i talks to u_j and v_x talks to u_y, where i, j, x, and y denote the relative positional ordering in the nodes, and either, $i < x < j < y$ or $i < x < y < j$ or $x < i < y < j$ or $x < i < j < y$ without loss of generality. In terms of an adjacency matrix, this can be regarded as if either the point (i, j) is in the lower left sub-matrix of (x, y) or vice versa, there is a crosstalk. Hence, our solution is to count the number of such occurrences. If this counting is done unintelligently, it can be in the order of $O(n^4)$. Our algorithm to compute the number of wire crossings runs in $O(n^2)$.

Figure 1.9 shown and example of wire crossing computation. The graph in Figure 1.9(a) can be represented by the adjacency matrix shown in Figure 1.9(b). The number of crossings in Figure 1.9(a) is 3. This



(a)

	1	2	3
A	1	1	0
B	0	0	1
C	1	0	0
D	0	1	0

(b)

	1	2	3
A	1	2	2
B	0	0	1
C	1	1	1
D	0	1	1

(c)

	1	2	3
A	2	4	5
B	1	2	3
C	1	2	2
D	0	1	1

(d)

Figure 1.9. Illustration of incremental wire crossing computation. (a) a bipartite graph with 3 wire crossings, (b) adjacency matrix of (a), (c) row-wise sum of (b) from left to right, (d) column-wise sum of (c) from bottom to top. Each entry in (d) now represent the total sum of entries in low-left sub-matrix. Using (b) and (d), wire crossing is $A2 \times B1 + B3 \times C2 = 3$, where the first entry is from (b) and second from (d).

can be obtained from the matrix by adding the product of every matrix element and the sum of its left lower matrix elements. i.e. the number of crossings is $\sum(A_{ij} \times \sum\sum A_{xy})$, where $i+1 < x < n$ and $1 < y < j-1$. This formula gives a good intuition of the process but is computationally very expensive. We now illustrate our method to calculate wire crossing more efficiently. First we take the row-wise sum of all entries as in Figure 1.9(c). Then we use this to compute the column-wise sum as in 1.9(d). Finally, we multiply all the entries in the original matrix and the column-wise sum matrix to compute the total wire crossing–each entry (r, c) in the original matrix is multiplied by the entry $(r+1, c-1)$ in the column-wise sum matrix as shown in 1.9(d). In the simulated annealing process, when we swap two nodes, it is identical to swapping the corresponding rows in the above matrices. Hence, it is enough if we just update the values of the rows in between the two rows that are being swapped. The pseudo-code for this incremental algorithm is as follows.

```
**calc_wire_crossing**($R_1, R_2, M$)
    if ($R_2 < R_1$)
        return(calc_wire_crossing($R_2, R_1, M$));
    $sum = pos = neg = diff = j = 0$;
    while ($j < NumRows$)
        $tmp = diff$;
        $i = R_2 - 1$;
        while ($i > R_1$)
            $sum = sum + M[i][j] * (pos - neg)$;
            $diff = diff + M[i][j];$
            $i = i + 1;$
        $sum = sum - M[R_1][j] * (tmp + neg);$
        $sum = sum + M[R_2][j] * (tmp + pos);$
        $pos = pos + M[i][j];$
        $neg = neg + M[R_2][j];$
    return(sum);
```

During cell placement, a move is done by randomly choosing a level in the graph and then swapping two randomly chosen gates $[g_1, g_2]$ in that level in order to minimize the total wirelength and wire crossing. In our implementation, the initial calculation of the wire length takes $O(n)$ and updating wire crossing takes $O(n^2)$ where n is the number of nodes in a layer of the bipartite graph. In our approach, we initially compute the wirelength and wire crossing and incrementally update these values after each move so that the update can be done much faster as illustrated above. This speedup allows us to explore a greater number of candidate solutions, and as a result, obtain better quality solutions. We set the initial temperature such that roughly 50% of the bad moves were accepted. The final temperature was chosen such that less than 5% of the moves were accepted. We used three different cost functions. The first cost function only optimized based on the net wirelength. The second cost function evaluated the number of wire crossings, while the last cost function looked at a weighted combination of both. The weights used were the ratio between the wirelength and the number of wire crossings obtained in the analytical solution.

6. Experimental Results

Our algorithms were implemented in C++/STL, compiled with gcc v2.96 run on Pentium III 746 MHz machine. The benchmark set consists of seven biggest circuits from ISCAS89 and five biggest circuits from ITC99 suites due to the availability of signal flow information.

Zone Partitioning Results

Table 1.2 shows the zone partition results for our QCA placement. The number of partitions is determined such there are 100 ± 10 majority gates per partition. We set the capacity of each wire block to 200 QCA cells. We compare the metrics and result of acyclic FM and QCA zone partitioning. The cost objective differs in the two comparisons of QCA partitioning ($\alpha = 1$ and $\beta = 0$) and acyclic FM ($\alpha = 0$ and $\beta = 1$), where α is related to wire block minimization and β is related to cutsize. With QCA partition, we see a 20% improvement in cutsize at the cost of 6%

Table 1.2. QCA zone partitioning results.

	Acyclic FM			Zone Partitioner		
name	cut	white	wire	cut	white	wire
b14	2948	151	138	2566	168	127
b15	4839	220	260	4119	144	256
b17	16092	1565	1789	13869	1616	1710
b20	6590	641	519	6033	642	518
b21	6672	599	560	6141	622	557
b22	9473	1146	1097	8518	1158	1098
s13207	2708	143	138	1541	144	137
s15850	3023	257	183	2029	254	181
s35932	7371	875	1014	5361	734	1035
s38417	9375	757	784	5868	775	773
s38584	9940	1319	1155	7139	1307	1095
s5378	1206	34	30	866	34	30
s9234	1903	99	81	1419	104	76
Ave	6318	600	596	5036	592	584
Ratio	1	1	1	0.8	0.99	0.98
time	14646			14509		

increase in runtime. A new algorithm was implemented, to reduce the number of white nodes, by taking into account terminal propagation. The initial gain of the affected cells is incremented twice. Our new algorithm for reducing the number of white nodes involves moving wire blocks to balance the variation in number of partitions per clocking level. Although our algorithm results in a 67% decrease in wire nodes and 66% decrease in white nodes, there is a tradeoff in a resulting increase in the number of wire crossings. Since wire crossing has been seen as a much more significant problem, we choose to sacrifice an increase in area for a decrease in the number of wire crossings.

Zone Placement Results

Table 1.3 details our zone placement results, where we report placement area, wirelength, and wire crossings for the benchmarked circuits. We compare the analytical solution to simulated annealing. Comparing simulated annealing to the analytical solution, we see an 87% decrease in wirelength and slight increase in wire crossings.

Cell Placement Results

Table 1.4 shows our cell placement results where we report net wirelength and number of wire crossings for the circuits using our analytical

Table 1.3. QCA zone placement results.

		Analytical		SA-based	
name	area	length	xing	length	xing
b14	20x17	81	67	23	67
b15	20x24	59	90	34	90
b17	69x52	3014	346	305	345
b20	36x36	414	165	99	166
b21	36x37	140	172	100	172
b22	48x50	1091	230	188	230
s13207	18x21	28	9	28	9
s15850	24x23	81	16	11	14
s35932	45x44	1313	64	78	68
s38417	42x43	493	54	48	54
s38584	55x48	1500	102	110	80
s5378	10x10	3	10	2	9
s9234	15x16	15	11	5	11
Ave		633	103	79	101
Ratio		1	1	0.13	0.98
time		23		661	

solution and all three flavors of our simulated annealing algorithm. We further tried simulated annealing from analytical start, and the results were identical to analytical solution. We observe in general that analytical solution is better than all three flavors of the Simulated Annealing methods, except in terms of wirelength in the case of the weighted Simulated Annealing process. But, the tradeoff in wire crossings makes the analytical solution more viable, since wire crossings pose a bigger barrier than wirelength in QCA architecture.

One interesting note is that when comparing amongst the three flavors of simulated annealing we find that simulated annealing with wire crossing minimization alone has the best wire crossing number, but surprisingly, in terms of wirelength, the simulated annealing procedure with wirelength alone as the cost function is not as good as the simulated annealing procedure which optimizes both wirelength and wire crossing. We speculate that this behavior is because lower number of wire crossings has a strong influence on wirelength, but smaller wirelength does not necessarily dictate lower number of crossings in our circuits.

7. Conclusions and Ongoing Works

In this article, we proposed a QCA partitioning and placement problem and present an algorithm that will help automate the process of design within the constraints imposed by physical scientists. Work to

Table 1.4. QCA cell placement results.

	Analytical		SA+WL		SA+WC		SA+WL+WC	
	wire	xing	wire	xing	wire	xing	wire	xing
b14	5586	1238	28680	23430	54510	3740	5113	4948
b15	9571	1667	23580	40400	69030	7420	8017	8947
s13207	3119	548	14060	15530	30610	1450	3250	1982
s15850	3507	634	18610	22130	42700	2140	3919	2978
s38417	9414	1195	45830	48400	80240	7320	9819	9929
s38584	19582	4017	59220	75590	140130	9820	20101	33122
s5378	1199	156	6280	6690	13600	730	1344	841
s9234	2170	205	10720	11540	23290	980	1640	2159
Ave	4192	741	16980	19950	38950	2740	3880	6878
Ratio	1	1	4.05	26.9	9.29	3.69	0.92	9.27
time	180		604		11280		12901	

address QCA routing and node duplication for wire crossing minimization are underway. Our ongoing work for zone placement includes a 2D placement solution, where the partitions are placed anywhere in the grid with the help of properly clocked routing channels. The outputs from this work and the work discussed here will be used to generate computationally interesting and optimized designs for experiments by QCA physical scientists. Finally, this work is an example of how systems-level research can positively affect physical device development - and why we should integrate both veins of research. Lastly, during this work it became apparent that a better picture of the QCA circuit design could be painted if we could compare the results from QCA placement to the placement of a CMOS circuit with the same functionality, and our ongoing work focuses on this issue.

References

Amlani, I, Orlov, A, Snider, G, and Lent, C (1998). Demonstation of a func. quantum-dot cellular automata cell. *J. Vac. Sci. Technology*, pages 3795–3799.

Amlani, I., Orlov, A., Toth, G., Bernstein, G., Lent, C., and Snider, G. (1999). Digital logic gate using quantum-dot cellular automata. *Science*, pages 289–291.

Bernstein, G (2003). Quantum-dot cellular automata: Computing by polarized systems. In *Proc. ACM Design Automation Conf.*

Cong, J. and Lim, S. K. (2000). Performance driven multiway partitioning. In *Proc. Asia and South Pacific Design Automation Conf.*, pages 441–446.

Gergel, N, Craft, S, and Lach, J (2003). Modeling qca for area minimization in logic synthesis. In *Proc. Great Lakes Symposum on VLSI*, pages 60–63.

Hamilton, S. (1999). Taking moore's law into the next century. *IEEE Computer*, pages 43–48.

Hennessy, K. and Lent, C. (2001). Clocking of molecular quantum-dot cellular automata. *Journal of Vacuum Science and Technology*, pages 1752–1755.

Ho, R., Mai, K., and Horowitz, M. (2001). The future of wires. *Proceedings of the IEEE*, pages 490–504.

Kummamuru, R., Timler, J., Toth, G., Lent, C., Ramasubramaniam, R., Orlov, A., and Bernstein, G. (2002). Power gain in a quantum-dot cellular automata latch. *Applied Physics Letters*, pages 1332–1334.

Lieberman, M., Chellamma, S., Varughese, B., Wang, Y., Lent, C., Bernstein, G., Snider, G., and Peiris, F. (2002). Quantum-dot cellular automata at a molecular scale. *Annals of the New York Academy of Science*, pages 225–239.

Mead, Carver and Conway, Lynn (1980). *Introduction to VLSI Systems.* Addison-Wesley Publishing Company.

Nguyen, J., Ravichandran, R., Lim, S. K., and Niemier, M. (2003). Global placement for quantum-dot cellular automata based circuits. Technical Report GIT-CERCS-03-20, Georgia Institute of Technology.

Niemier, M. and Kogge, P. (2001). Exploring and exploiting wire-level pipelining in emerging technologies. In *Proc. Great Lakes Symposum on VLSI*, page Int. Conf. on Computer Architecture.

Niemier, M. and Kogge, P. (2004). The 4-diamond circuit: A minimally complex nano-scale computational building block in qca. In *IEEE Sym. on VLSI*, pages 3–10.

Niemier, Mike (2003). The effects of a new technology on the design, organization, and architectures of computing systems. PhD Dissertation, Univ. of Notre Dame.

Packan, P. (1999). Pushing the limits. *Science*, pages 2079–2081.

Rabaey, J. M. (1996). *Digital Integrated Circuits: A Design Perspective.* Prentice Hall Electronics.

Ravichandran, R., Ladiwala, N., Nguyen, J., Niemier, M., and Lim, S. K. (2004). Automatic cell placement for quantum-dot cellular automata. In *Proc. Great Lakes Symposum on VLSI.*

Rutten, P. (2001). Is moore's law infinite? the economics of moore's law. *Kellog Tech Venture*, pages 1–28.

Snider, G., Orlov, A., Amlani, I., Bernstein, G., Lent, C., Merz, J., and Porod, W. (1999). Quantum-dot cellular automata: Line and majority gate logic. *Jpn. J. of Applied Physics*, pages 7227–7229.

Sugiyama, K., Tagawa, S., and Toda, M. (1981). Methods for visual understanding of hierarchical system structures. *IEEE Trans. Syst. Man,. Cybern*, pages 109–125.

Tougaw, P. and Lent, C. (1994). Logical devices implemented using quantum cellular automata. *Journal of Applied Physics*, page 1818.