The Magic of a Via-Configurable Regular Fabric

Yajun Ran and Malgorzata Marek-Sadowska

Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106 USA

Abstract

In this paper we provide a comprehensive study of the mappability of a via-configurable gate array (VCGA). Although the base cell of the VCGA is simple, by customizing only via masks it can implement various combinational logic functions, sequential elements, and SRAM cells. Our VCGA can be efficiently configured into SRAM arrays, adders and multipliers. The strong configurability of our VCGA allows us to minimize the number of fixed parts in a general-purpose VCGA fabric, which greatly improves area utilization.

1 Introduction

Due to huge performance and cost gaps between cell-based and *FPGA*-based designs, *structured-ASICs* have become the preferred option for many applications. Structured-ASICs have most of their parts pre-fabricated, and designers have flexibility for implementing different circuits in the remaining metallization steps. Among various structured-*ASIC*s, via-configurable fabrics, which use only via masks to program circuit functionality, have been favored due to reduced mask writing effort.

In [2], Patel *et al* proposed a via-patterned gate array (*VPGA*), whose architecture follows an *FPGA* in which active switches and programming bits are replaced by potential vias. In [3], Pileggi *et al* proposed a hybrid block structure, which consists of a 3-*LUT* and several *NAND* gates. In [1], Hu *et al* proposed a semi-universal logic block constructed from several simple gates. All of these structures have fixed basic gates (or *LUT*s) and use vias to provide connections between them. In [4], we proposed a novel, via-configurable cell structure which has fixed layout patterns and uses vias to implement cell functionality. Our fabric, which is structurally similar to standard cells, can achieve performance and area comparable to the cell-based designs.

In general, a fabric with many pre-fabricated parts is less flexible than cell-based designs. For certain specific application needs, conventional fabrics introduce some fixed parts, such as datapath elements and memory blocks. The obvious drawback is that different applications may have totally different requirements. As a result, fixed parts could cause large area waste. In this paper, we show that although our *VCGA* has a simple base-cell structure, it is highly flexible and efficiently implements many practical functions, including combinational logic, sequential logic, *SRAM* cell, and various arithmetic units. Moreover, all the configurations can be completed merely by customizing a set of via masks. The high flexibility of our *VCGA* makes it a promising general-purpose fabric.

2 Base Cell Structure

The basic logic element (*BLE*) of our *VCGA* consists of a via-configurable functional cell (*VCC*) and two neighboring inverter arrays. A *VCC* is composed of vertically aligned transistor pairs and single n-/p-diffusion strips. *M1* segments are placed vertically, and *M2* segments are placed horizontally. Figure 1 shows a stick diagram of a 5-*VCC*, which contains five transistor pairs and is the base cell of our *VCGA*. The intersections between *M2* segments and *M1* segments are potential via sites. Cell functionality is implemented by appropriate via configurations. When *M2* segments W_1–W_8 are not used for cell customization, they can be used for inter-cell routing. The *M1* segments C_1 and C_2 connect N-part and P-part together for a static *CMOS* gate. Moreover, they provide the feedback to *VCC* inputs through two *M2* segments F_1 and F_2. F_1 and F_2 are also used when several *VCC* inputs connect to the same signal.

Figure 2 shows a stick diagram of an inverter array. It can be configured as four independent inverters, two 2X inverters, or a single 4X inverter, etc.

Four *BLE*s form a via-configurable block (*VCB*). In each *VCB*, the *VCC*s are rotated by 90 degrees with respect to their neighboring *VCC*s. Between any two neighboring *VCC*s, there is an inverter array which provides the options of inverting outputs of the ascendant *VCC*, or inverting inputs of the descendant *VCC*. A *VCGA* is an array of *VCB*s. Figure 3 shows a 4x4 *VCGA* array. Horizontal and vertical *M2* wire segments between *BLE*s provide the connections between neighbors. Each segment spans two *BLE*s, and the

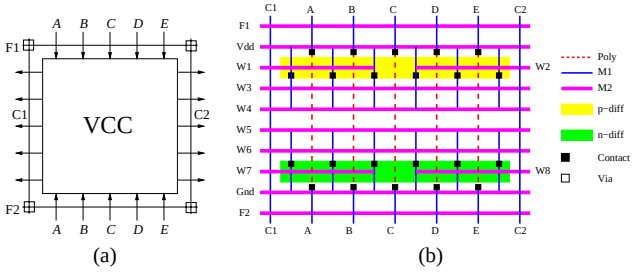


Figure 1. Stick diagram of the base 5-*VCC* cell

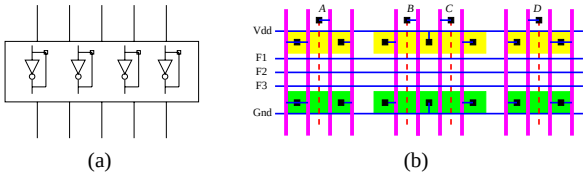


Figure 2. Stick diagram of the base inverter array

segments are distributed in a staggered way. *M1* jumpers provide the connections between two *M2* segments in the same row (column). They are also used for connections between horizontal and vertical segments. Upper-level wire segments, which are properly segmented, provide the long-distance connections between *BLE*s.

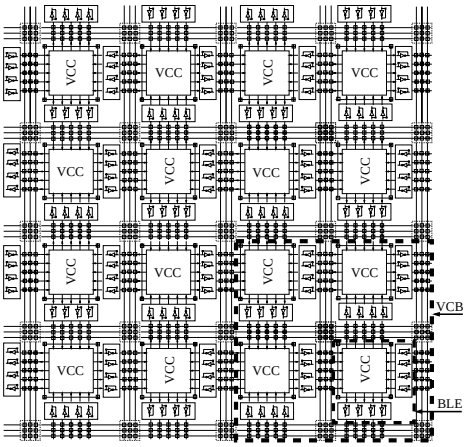


Figure 3. A 4x4 *VCGA* array

3 Combinational Logic

We first show the capability of our *VCC* and inverter array to implement different combinational functions. In the following, all our experimental data are obtained for $0.18\mu m$ technology parameters with $V_{dd} = 1.8V$. The areas of a 5-*VCC*, an inverter array and a *VCB* are 58, 54 and 800 μm^2, respectively.

3.1 Single Logic Gate and Multi-Gate Implementation

As shown in [4], a 5-*VCC* can implement about 92% of logic functions using five transistor pairs. A *VCC* can also be configured to implement two gates by letting them share V_{dd} and ground in the middle. Figure 4 shows a 5-*VCC* implementing an *XOR* gate, where $\bar{A}$ and $\bar{B}$ are supplied by the *VCC*-feeding inverter array. An *XOR* gate needs only four transistor pairs, and thus one transistor pair is unused. Figure 5 shows a 5-*VCC* implementing two gates. Flexibility of the *VCC*-cell significantly contributes to high cell utilization. Due to the similar structure, a *VCC*-implemented logic gate has similar performance as a standard cell but is about 50% larger.

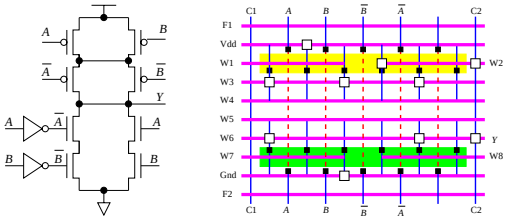


Figure 4. Implementation of an *XOR* gate $Y = A \oplus B = \overline{A\bar{B} + A\bar{B}}$ by a 5-*VCC*

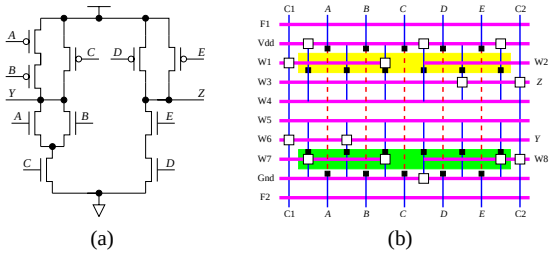
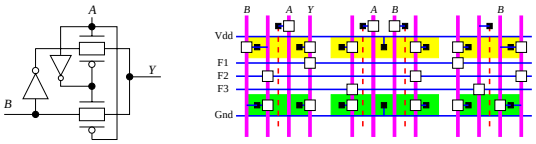


Figure 5. Implementation of two gates $Y = \overline{(A+B)C}$ and $Z = \overline{DE}$ by a 5-*VCC*

The inverter array can also be configured to implement some logic functions. Figure 6 shows an *XOR* gate built from the inverter array by using pass-transistor logic. Table 1 lists the characteristics of two *XOR* implementations. T_{wc} is the worst-case delay, and T_{avg} is the average delay among all input combinations. The pass-transistor implementation by inverter array has better performance than the static *CMOS* implementation by *VCC*. The *VCC*-implemented *XOR* gate could be used on non-critical path to balance the cell utilization.

Similarly, a 2-to-1 *MUX* can also be configured from the inverter array as shown in Figure 7. With appropriate via configurations, some 2-input functions can be implemented by a 2-to-1 *MUX* as shown in [4].



(a) (b)

Figure 6. Implementation of an *XOR* gate $Y = A \oplus B = \bar{A}B + A\bar{B}$ by an inverter array

Table 1. Comparison of two *XOR* implementations: input slew = 150ps, load capacitance = 25 fF.

Style	Area (μm^2)	T_{wc} (ps)	T_{avg} (ps)
5-*VCC* + 2 INV	85	361	270
inverter array	54	248	147

3.2 Repeaters

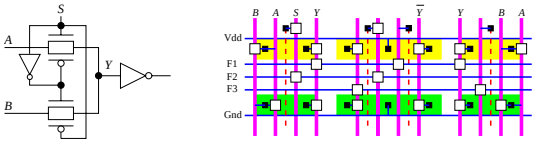
The *VCC* and the inverter array can be configured to form inverters of different sizes. Repeaters, which play an important role in interconnect performance optimization, can be dynamically configured from the *VCGA* elements. For example, a *BLE* can provide inverters 1X–13X-size of the minimum, and a *VCB* can be configured into an inverter of up to 52X of the minimum size with inter-*BLE* connections.

3.3 Capacitors

A *VCC* (inverter array) can also be configured to be a capacitor, which may be used as a decoupling component to reduce power/ground voltage fluctuations. One implementation is to connect all the transistor gates together to form one plate, and to connect all the drains/sources together to form another plate of a capacitor. In this way a *BLE* in the *VCGA* can provide capacitance of 44 fF.

4 Sequential Elements

Sequential elements can be constructed from basic gates. With the ability to implement two gates, and using the feedback segments C_1–C_2/F_1–F_2, a *VCC* can implement sequential elements.

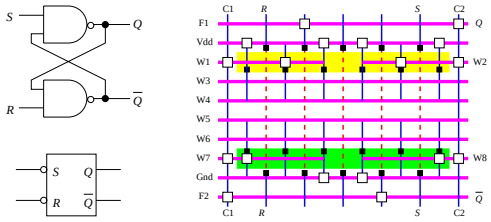


(a) (b)

Figure 7. Implementation of *MUX21* $Y = SA + \bar{S}B$ by an inverter array

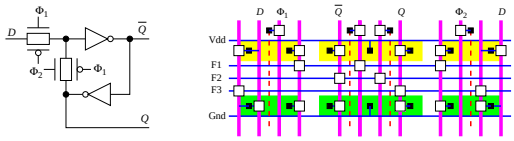
4.1 Latch and Flip-Flop (FF)

Figure 8 shows an *RS* latch implemented by a 5-*VCC*. Figure 9 shows a D-latch implemented by the inverter array, where Φ_1 and Φ_2 are two non-overlapping clocks. A flip-flop can be constructed using several *VCC*s (inverter arrays). Therefore, no fixed sequential elements are required in our fabric.



(a) (b)

Figure 8. Implementation of an *RS* latch by a 5-*VCC*



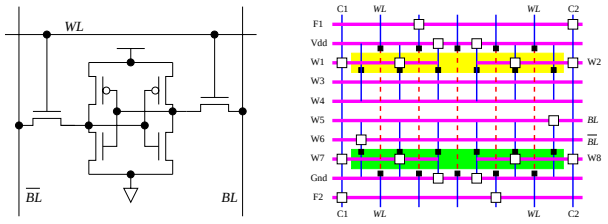
(a) (b)

Figure 9. Implementation of a D-latch by an inverter array

4.2 SRAM

A *VCC* can also be configured to be an *SRAM* cell. Figure 10 shows the configuration, where the middle transistor pair and the two PMOS transistors for word-line *WL* are not used. Moreover, an inverter array can also be configured to be an *SRAM* cell as shown in Figure 11. As a result, each *BLE* can implement a 3-bit memory. A large memory array can be constructed from our *VCGA*. Figure 12 shows a 6-bit memory formed by two *BLE*s with the corresponding *M2* and *M3* wire organizations.

Our *VCGA* can be configured into a memory array with 15K-bit/mm^2. Although the memory array constructed in this way is less dense than a customized one (∼90K-bit/mm^2), it provides the flexibility to build memory as needed, both in size and in location. For a fabric used for various applications, this flexibility could save unnecessary area waste caused by a fixed-size memory block. It is possible to combine area efficiency and design flexibility by having in the fabric a customized memory block suiting a class of applications, and to provide the supplementary memory using dynamic configuration.



(a) (b)

Figure 10. Implementation of a 6-transistor *SRAM* cell by a 5-*VCC*

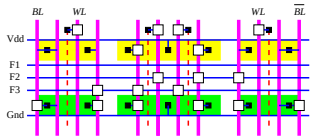


Figure 11. Implementation of a 6-transistor *SRAM* cell by an inverter array

5 Arithmetic Units

In this section, we show how to construct some arithmetic units for building datapath using our *VCGA*.

5.1 Full Adder (FA)

The most common arithmetic unit is a full adder. Given two adder inputs a and b, and a carry-in c_i, the sum s and the carry-out c_o can be expressed as $s = a \oplus b \oplus c_i$ and $c_o = ab + (a + b)c_i$, respectively. Figure 13 illustrates the implementation of a full adder by a *BLE*. The carry-out c_o is implemented by configuring a 5-*VCC* into a 5-input complex gate. The signal s is implemented by configuring two inverter arrays into two *XOR* gates (see Figure 6).

5.2 Ripple Carry Adder (RCA)

Figure 14 shows a 16-bit *RCA*, where each dashed box corresponds to a full adder. The zig-zag structure of the ripple path is caused by the orthogonal orientations of neighboring *VCC*s.

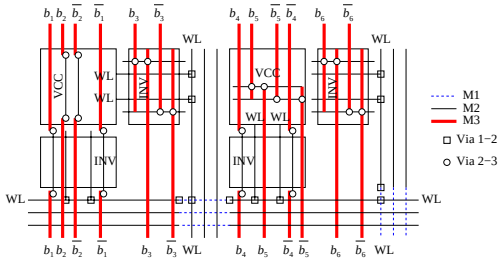


Figure 12. A 6-bit memory word constructed from a *VCGA*

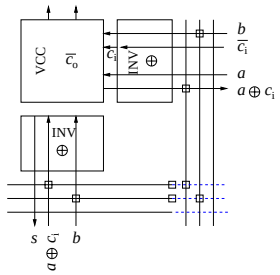


Figure 13. A full adder by a *BLE*

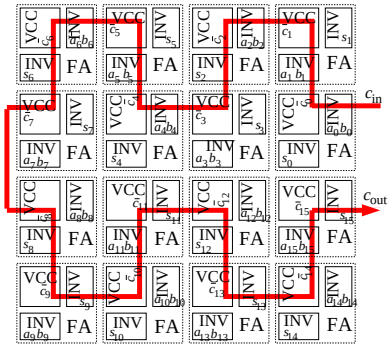


Figure 14. A 16-bit ripple carry adder, the bold line shows the carry ripple path.

5.3 Carry-Lookahead Adder (CLA)

A carry-lookahead adder is one of the most frequently used fast-addition adders. It is based on the idea of dividing inputs into k-bit groups and organizing them into a tree-like structure. The carries for each group are obtained from the carry lookahead tree, not by waiting for the carry rippling from the least significant bit. Figure 15 shows an example of a 16-bit *CLA* with $k = 3$.

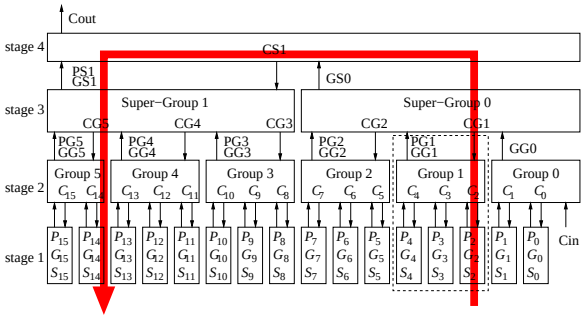


Figure 15. A 16-bit *CLA*, the bold line shows the critical path.

Given inputs $A_{<15..0>}$, $B_{<15..0>}$, and carry-in C_{in}, the local carry-generate G_i, carry-propagate P_i, and sum S_i in stage 1 can be expressed as $G_i = A_iB_i$, $P_i = A_i \oplus B_i$ and $S_i = P_i \oplus C_i$, where C_i is the carry-in for the i-th bit,

which comes from the lookahead circuitry. In a *CLA*, the group (super-group) carry generate/propagate and carry C_i signals have the similar functional forms. For example, in Figure 15,

$$GG_1 = G_4 + P_4(G_3 + P_3G_2) \quad (1)$$
$$PG_1 = P_4P_3P_2 \quad (2)$$
$$GS_1 = GG_5 + PG_5(GG_4 + PG_4GG_3) \quad (3)$$
$$PS_1 = PG_5PG_4PG_3 \quad (4)$$
$$CG_4 = GG_3 + PG_3CS_1 \quad (5)$$
$$C_2 = CG_1 \quad (6)$$
$$C_3 = G_2 + P_2CG_1 \quad (7)$$
$$C_4 = G_3 + P_3(G_2 + P_2CG_1). \quad (8)$$

Carry generate signals (also carry signals C_i) have two forms, a fanin-3 gate (e.g., (5) and (7)) and a fanin-5 gate (e.g., (1), (3) and (8)). A 5-*VCC* can implement a fanin-5 gate. A 5-*VCC* can also implement a fanin-3 gate, leaving an empty site for another 2-input gate. Carry propagate signals are formed by a 3-input *AND* gate (e.g., (2) and (4)), which can be implemented by part of a 5-*VCC*.

Figure 16 shows the organization of a *VCGA* implementing the dashed block in Figure 15. The $P_i/G_i/S_i$ module in stage 1 is implemented by a *BLE* with an empty site for a 3-input gate. The arrowed lines show connections between those blocks. *Site-2* and *Site-3* correspond to the empty sites which can be used for other purposes, e.g., folding the top-level of the carry-lookahead tree. With careful planning, a compact *CLA* can be constructed.

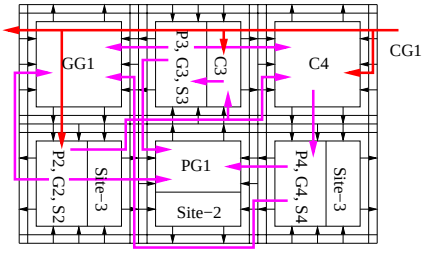


Figure 16. A *CLA* block implemented by a *VCGA*

5.4 Multiplier

Figure 17 shows an 8x8 multiplier ($P = X \cdot Y$) which uses radix-4 multiplication scheme (Booth's coding) [5]. It consists of three main parts: a Booth encoder, a carry-save-adder (*CSA*) array and a final carry-propagate-adder (*CPA*). A Booth encoder generates three control values for each partial product: *ZERO* which zeroes the operand, *NEG* which negates the operand, and *TWO* which doubles the operand (left shift by one bit). The accumulation of partial product is implemented by a *CSA* array. The final *CPA* generates the final sum. Figure 18(a) shows the Booth encoder cell, and Figure 18(b) shows the Booth-encoded multiplier cell. A Booth encoder cell can be implemented by four *BLE*s. A Booth-encoded multiplier cell consists of a Booth *MUX* and a *CSA* cell (a full adder). It can be implemented by two *BLE*s, one for the full adder and the other for the Booth *MUX* (an inverter array for the 2-to-1 *MUX*, the other inverter array for the *XNOR* gate, and a *VCC* for the *NOR* gate).

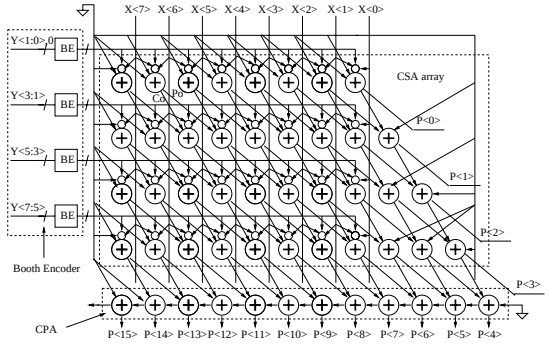
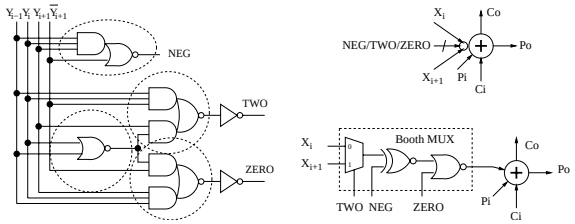


Figure 17. A 8X8 radix-4 Multiplier



(a) Booth encoder (b) Multiplier cell

Figure 18. Booth encoder and multiplier cell

Figure 19 illustrates the organization of a *VCGA*-implemented *CSA* array. The multiplier cells (each implemented by two *BLE*s) are interleaved to make use of inter-*BLE* direct connections coming from the orthogonal orientation of neighbors in our *VCGA*. The connections between multiplier cells are implemented by inter-*BLE* wire segments. The Booth coding signals using *M3* are on top of the cells.

6 Experiments

We synthesized into *VCGA* a 32-bit *RCA*, a 32-bit *CLA* and a 16X16-bit multiplier (using a *CLA* as a final *CPA*). Table 2 lists the characteristics of those components. For comparison, we also manually mapped those circuits into a 3-*LUT*-based fabric (to mimic the *VPGA* in [2]) and a hybrid-block-based fabric [3]. The hybrid block contains one 3-*LUT*, two 3-input *NAND* gates, and seven inverters. Due to the difficulty of implementing sequential elements by *LUT*s, in their original version both fabrics have a fixed

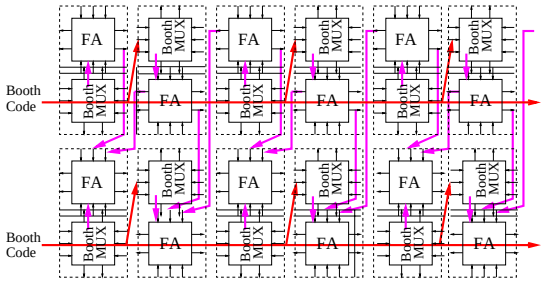


Figure 19. Portion of a *CSA* array by a *VCGA*

DFF in each block, that cannot be used in these arithmetic units. Moreover, many inverters in the hybrid block are not used. The fixed *DFF* and inverters cause a significant area waste in the implementation of arithmetic components. For more aggressive comparisons, we assume that there are no *DFF*s in those blocks, and inverters are assigned as required (hence no inverters are wasted) for those two fabrics. In the following we only compare the three fabrics in terms of logic area and assume that inter-*LUT*(block) connections use customized wires on the top of transistors. The same fixed routing structure as in [3] could be applied on them. Due to the smaller logic area requirement of *VCGA*-based designs, more routing resource could be available to relieve routability issue.

Table 2. Arithmetic units comparison by different fabric implementations (Power consumption is calculated at 100MHz).

Circuit	Type	Area (μm^2)	Delay (ns)	Power (mW)	Utilization (%)
32-bit RCA	VCGA	5568	6.39	0.84	100
	3-LUT	4672	8.64	1.37	100
	Hybrid	6784	8.64	1.42	93.8
32-bit CLA	VCGA	11484	1.68	1.16	78.3
	3-LUT	14016	3.24	4.11	66.7
	Hybrid	12879	2.14	2.49	75.5
16x16-bit Multiplier	VCGA	58812	5.73	7.49	81.7
	3-LUT	51246	8.91	15.0	93.2
	Hybrid	72398	7.88	13.1	80.5

From Table 2 we see that *VCGA*-based designs have significant advantages in delay and power consumption over purely 3-*LUT*-based and hybrid-block-based fabrics. We attribute the superior results of *VCGA*-based designs to more efficient implementation of functions using single-stage static *CMOS* gates, rather than 3-*LUT*s. 3-*LUT*-based designs show area advantage in *RCA* implementation, since each *LUT* is compact due to customized design, and two 3-*LUT*s are a perfect match for a full adder. Each multiplier cell in a *CSA* array also can be perfectly realized by four 3-*LUT*s, which results in a compact area of 3-*LUT*-based multiplier. The configuration flexibility of our *VCC* and inverter array invokes an area penalty compared to a customized layout. However, the strong configurability helps to achieve high transistor utilization as shown in the column *Utilization*. When a 3-*LUT*-based fabric does not match circuit structure well, for example, in CLA implementation, it incurs a large penalty in every aspect. Moreover, if we considered the fixed *DFF*s and inverters, the transistor utilization of the designs implemented by *LUT*s (either purely-*LUT*-based or hybrid-block-based) in Table 2 will decrease significantly; therefore the area of the designs would increase a lot (could be doubled due to the large area of an *FF*). The hybrid-block-based fabric has delay and power advantages over purely 3-*LUT*-based fabric since a *NAND3* gate is faster and less power-consuming than a 3-*LUT*. But the block is more ad-hoc and irregular. Furthermore, a *NAND3* gate is not efficient to implement some complex gates, such as a fanin-5 gate, an *XOR* gate etc, which makes the arithmetic units implemented by hybrid-block-based fabric slower and more power-consuming compared to *VCGA*-based designs.

7 Conclusions

In this paper we have shown the strong configurability of a via-configurable gate array. Using only via masks, a *VCGA* can efficiently implement combinational logic, sequential elements, repeaters, *SRAM* blocks, and datapath elements. It has much better configurability and performance than *LUT*-based fabrics. Our plan for future work includes the routing structure design and CAD support for this fabric.

8 Acknowledgments

This work was supported by NSF through CCR grant 0098069. The authors also acknowledge the Intel equipment grant.

References

[1] B. Hu, H. Jiang, Q. Liu, and M. Marek-Sadowska. Synthesis and placement flow for gain-based programmable regular fabrics. In *Proceedings of International Symposium on Physical Design*, pages 197–203, 2003.

[2] C. Patel, A. Cozzie, H. Schmit, and L. Pileggi. An architectural exploration of via patterned gate arrays. In *Proceedings of International Symposium on Physical Design*, pages 184–189, 2003.

[3] L. Pileggi et al. Exploring regular fabrics to optimize the performance-cost trade-off. In *Proceedings of Design Automation Conference*, pages 782–787, 2003.

[4] Y. Ran and M. Marek-Sadowska. On designing via-configurable cell blocks for regular fabrics. In *Proceedings of Design Automation Conference*, pages 198–203, 2004.

[5] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, 1993.