Theory Comput. Systems **38**, 503–536 (2005)
DOI: 10.1007/s00224-004-1196-9

Theory of Computing Systems

Efficient Data Storage in Large Nanoarrays*

Lee-Ad J. Gottlieb, John E. Savage, and Arkady Yerukhimovich

Department of Computer Science, Brown University,
Providence, RI 02912-1910, USA
{legottli, jes}@cs.brown.edu
arkady@alumni.brown.edu

Abstract. We explore the storage of data in very large crossbars with dimensions measured in nanometers (nanoarrays) when h-hot addressing is used to bridge the nano/micro gap. In h-hot addressing h of b micro-level wires are used to address a single nanowire. Proposed nanotechnologies allow subarrays of 1s (stores) or 0s (restores) to be written. When stores and restores are used, we show exponential reductions in programming time for prototypical problems over stores alone. Under both operations, it is **NP**-hard to find optimal array programs. Under stores alone it is **NP**-hard to find good approximations to this problem, a question that is open when restores are allowed. Because of the difficulty of programming multiple rows at once, we explore the programming of single rows under h-hot addressing. We also identify conditions under which good approximations to these problems exist.

1. Introduction

The end of photolithography as the driver for Moore's Law is predicted within 8 to 13 years [1]. Although this might be seen as an ominous development, nanotechnologies are emerging that are expected to continue the technological revolution. One of the most promising nanotechnologies is the crossbar [17], [23], a two-dimensional array (**nanoarray**) formed by the intersection of two orthogonal sets of parallel and uniformly spaced nanometer-sized wires, **nanowires**, as suggested in Figure 1(a). Nanowires can be built from carbon nanotubes [9] and semiconducting materials [5], [21]. Experiments have shown that nanowires can be aligned in parallel with nanometer spacings using a form of directed self-assembly (the dimensions are too small for photolithography to be

* This research was funded in part by NSF Grant CCR-0210225.

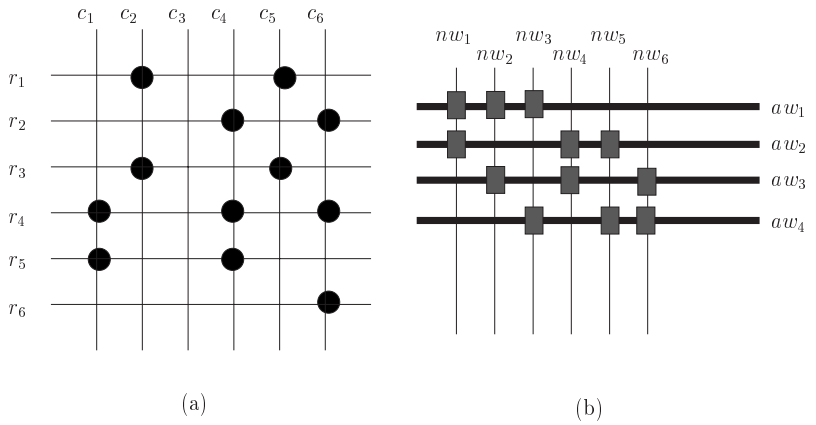


Fig. 1. (a) A nanoarray and (b) 2-hot addressing of six nanowires $\{nw_1, \ldots, nw_6\}$ with four micro-level address wires $\{aw_1, \ldots, aw_4\}$. In practice the microwires would be much larger than the nanowires.

used) and that two such layers of nanowires can be assembled at right angles [14], [20]. Materials have been developed that permit nanoarrays to store binary data [2], [4], [22] at **crosspoints**, the intersection of a pair of orthogonal nanowires. Thus, nanoarrays for data storage are a realistic possibility. (See Section 2.)

Nanoarrays offer both an opportunity and a challenge. The opportunity is to store enormous quantities of data. (A density of 10^{11} crosspoints per cm^2 has been achieved [20].) The challenge is to enter the data efficiently. We address the latter problem in this paper. If the time to enter data into big nanoarrays is very large, their use may be limited, especially if their contents change frequently, as in field programmable gate arrays. However, given the very large storage capacities envisioned, reductions in programming time by constant factors may be important.

The nanoarray technologies that have been proposed by Kuekes et al. [17] and Rueckes et al. [23] allow the crosspoints between two orthogonal nanowires to be made conducting (denoted by 1) or non-conducting (denoted by 0) by applying a large positive or negative voltage between them. The technologies also allow subarrays of 1s (*store*s) or 0s (*restores*) to be written at the intersection of sets of rows and columns. We explore array programming using h-hot addressing, a technique in which each nanowire has b potentially active regions (each associated with an orthogonal microwire) exactly h of which must be "hot" (turned on) to address a nanowire [8], as suggested in Figure 1(b) and described in Section 3.

In this paper we consider the following questions: (a) "How efficiently can particular arrays be programmed?" (b) "How difficult is it to find a minimal or near minimal number of stores and restores to program an arbitrary array?"

In Section 3.3 we use counting arguments to develop lower bounds that apply to most array programming problems under h-hot addressing for $h \geq 1$. It is not surprising that most arrays require a large number of programming steps. However, it is very probable that particular arrays of interest are not among the arrays for which these lower bounds apply.

We address question (a) by identifying a set of structured prototypical arrays in Section 3.4 and develop efficient programs for many of them in Section 4. We find that when only stores are used, 1-hot programs for these prototypical arrays require a number of steps proportional to the number of rows and columns in the arrays. However, when restores are also used, dramatic reductions in the number of steps are possible. Under 1-hot addressing we show that diagonal $n \times n$ arrays can be programmed in $2\lceil \log_2 n \rceil$ steps (see Section 4.4). When h-hot addressing is used, the number of steps is no more than $2 \log_2 n + 2h$ (see Section 4.5). We also show that these bounds are optimal to within a factor of two by deriving lower bounds on the number of operations that is needed (see Section 4.7). We obtain similar results for lower-full and upper-full matrices (see Sections 4.8 and 4.9). Under 1-hot addressing, we show that banded arrays can be programmed in $6(\log n + 2)$ steps but that this number grows to $O\left(3^h(1 + \beta/n^{1-1/h})(1/h) \log_2 n\right)$, a function that grows exponentially in h (see Sections 4.10 and 4.11). Finally, we derive upper bounds on the number of steps to program s-sparse arrays, arrays that have at most s 1s in each row and column and show that $O\left(s^2 \log^2(n/s^2)\right)$ steps suffice under 1-hot addressing. No results have been obtained for this problem under h-hot addressing when $h \geq 2$.

We address question (b) by studying the complexity of array programming in Section 5. Through a series of steps we show that the ARRAY PROGRAMMING decision problem is **NP**-complete under stores and restores. The first of these steps is to show that the problem is equivalent to known problems from which we conclude that it is **NP**-complete under stores alone. In three additional steps we show that it is **NP**-complete under both stores and restores. We first show that it is **NP**-complete when the number of operations is logarithmic in the size of an array. This is done via a reduction from THREE EDGE COVERING. We then identify a specific array for which the unique optimal algorithm under stores and restores uses only stores and combine this information with the above result to complete the proof.

In Section 6 we explore the approximability of array programming and equivalent optimization problems. We show that under stores alone they are not approximable to within a factor of N^{ε} of optimal for some $\varepsilon > 0$ in polynomial time if $\mathbf{P} \neq \mathbf{NP}$ where N is the size of the problem. The question of approximability when stores and restores are used is open. The good news here is that under specialized conditions, some of which may prevail in practice, the array programming problem is log-approximable under stores alone. We show that this is true when (a) the number of 1s in rows or columns of an array is bounded, (b) the stores and restores are restricted to programming subarrays of bounded size, or (c) $h = b/2$, a case for which optimal algorithms may require many more steps than when h is small by comparison with b.

Because array programming with h-hot addressing is hard, we explore the complexity of programming arrays by single rows or columns. We show in Section 7 that h-hot row programming is **NP**-hard under stores and restores and, when only stores are used, is hard to approximate in polynomial time unless **P= NP**. As with array programming, the approximability question is open under stores and restores.

Because it is hard to program arrays, in Section 8 we explore heuristics for this problem under both stores and restores. Our approach assumes the existence of a good

heuristic to find the largest subarray of 1s or 0s in an $n \times n$ array, a challenging computational problem in its own right.

Conclusions and open problems are presented in Section 9.

2. Nanoarray Technologies

Kuekes et al. [17] propose placing a supramolecular layer between the vertical and horizontal sets of wires in a nanoarray. A supramolecule consists of a molecular chain on which a dumbbell-type ring is located. The position of the ring on the chain can be raised or lowered through the application of a large positive or negative electric field. In one position the supramolecule is conducting; in the other it is not. When a large positive electric field is placed between orthogonal nanowires, the supramolecules in the vicinity of their intersection enter one of two states; when a large negative electric field is applied, they enter the other state. Data is stored in the array in this fashion. The state of supramolecules in the vicinity of the intersection of two orthogonal nanowires can be sensed by applying a voltage that is large enough to detect whether or not current will flow but not so large that it changes the state of the supramolecules.

Rueckes et al. [23] have proposed a second technology based on carbon nanotubes to record one of two states at the intersection of two orthogonal wires. They suspend one carbon nanotube on insulators above a second nanotube. When a sufficiently large attractive electric force is applied, the two nanotubes come into contact and a current can flow. When the force is released, the van der Waals force maintains contact. This process can be reversed by applying a sufficiently large repulsive force. The new position, in which no current can flow, is maintained by the elastic force in the nanotube. Data is stored in the array by closing and opening contacts. Data can be sensed by applying a voltage that is large enough to produce a detectable current without changing the state of intersections.

Recently Nantero (see http://www.nantero.com) has announced that it "is developing . . . a high-density nonvolatile random access memory device using nanotechnology" that "involves the use of suspended nanotube junctions as memory bits." The device is said to have a total storage capacity of 10^{10} bits.

2.1. *Related Results*

Few results related to those presented here are known to the authors. This is likely due to the fact that data storage time has not been a limiting resource. However, the importance of data storage time has increased in the context of field-programmable gate arrays (FPGAs). In a pair of papers, Hauck and co-authors [11], [18] have explored configuration compression for the Xilinx XC6200 FPGA, a problem of increasing importance now that FPGAs are being dynamically reconfigured. Modules in FPGAs, each of which has a binary address, are configured by writing words into them. The Xilinx chip has a wildcard register that allows multiple modules to be configured at the same time. The set of binary addresses consistent with all values of the bits whose positions are 1 in the wildcard register are activated and the data word supplied as input is written into all modules with these addresses. Hauck et al. have modeled this use of wildcards as a logic minimization

problem, which is **NP**-hard. Using the Espresso heuristic logic minimizer [3] they show a reduction in the number of configuration steps by a factor of four to seven over the non-compressed method. While their model is related to our model, they are not the same. We allow arbitrary groups of horizontal and vertical lines to be addressed, which provides much more flexibility than they allow.

3. Array Programming

In this section we formally model the array programming problem when row and column wires in an array are either controlled individually or using h-hot addressing. We also derive lower bounds on the number of programming steps for most $n \times n$ binary arrays when $h = 1, 2, 3$ and $h = O(\log n)$. Finally, we introduce structured array programming problems that are used in later sections to show that restores can have a dramatic effect on the number of operations to program arrays.

3.1. *h-Hot Addressing*

Definition 3.1. The ***h*-hot addressing scheme on *b* address wires**, $h < b$, assigns a unique weight-h binary b-tuple **a** to each integer in the set $N = \{1, 2, \ldots, n\}$, $n \leq \binom{b}{h}$. If a_j is the index of the jth 1 in **a**, then **a** is uniquely represented by the **address *h*-tuple** $(a_1, a_2, \ldots, a_h)$ of distinct integer **addresses** in the set $A = \{1, 2, \ldots, b\}$. Denote with $(a_1^{(i)}, a_2^{(i)}, \ldots, a_h^{(i)})$ the address h-tuple for integer i.

Address wire *aw*$_j$ is associated with address j in A. **Nanowire *nw*$_i$** is associated with integer i in N. (See Figure 1(b).) An address wire is either **active** or **inactive** and a nanowire is either "addressed" or "not addressed".

Nanowire nw_i is **addressed** if the set of activated address wires includes every address wire aw_j with $j \in \{a_1^{(i)}, a_2^{(i)}, \ldots, a_h^{(i)}\}$.

3.2. *Array Programming under h-Hot Addressing*

Definition 3.2. In an $n \times n$ array $\mathcal{W}$ let $R \subseteq \{1, 2, \ldots, n\}$ and $C \subseteq \{1, 2, \ldots, n\}$ be sets of rows and columns, respectively. A **store** (**restore**) is a single-step operation in which $w_{i,j} \in \mathcal{W}$ is set to 1 (0) for all $i \in R$ and $j \in C$. A **program** for an $n \times n$ array $\mathcal{W}$ is a sequence of store and restore operations that produces the $n \times n$ array $\mathcal{W}$ when applied to an $n \times n$ array whose entries are initially all 0.

We represent a store (restore) operation on an array $\mathcal{W}$ associated with rows $R \subseteq \{1, \ldots n\}$ and columns $C \subseteq \{1, \ldots, n\}$ by an $n \times n$ **(de)activation matrix** $A_{R,C} = \left[a_{i,j}^{R,C}\right]$ $\left(D_{R,C} = \left[d_{i,j}^{R,C}\right]\right)$ in which all entries are 0 (1) except those at the intersection of rows $i \in R$ and columns $j \in C$, which are 1 (0) where the rows and columns are addressed by h-hot addressing. The addressable sets of rows and columns are restricted by h-hot addressing.

The goal of the **array programming optimization problem** is to find the smallest number of operations needed to program a given binary $n \times n$ array when the entries in the array are initially 0.

The following decision problem captures the essential difficulty of the h-hot array programming problem. Later we refer to AP which is the problem h-AP when $h = 1$.

h-HOT ARRAY PROGRAMMING(h-AP)
Instance: Four-tuples $(\mathcal{W}, h, b, k)$ where $\mathcal{W}$ is an $n \times n$ array over $\{0, 1\}$, $h \leq b$, and k are integers. The n rows and columns are addressed using h-hot addressing with b address lines where $n \leq \binom{b}{h}$.
Answer: "Yes" if $\mathcal{W}$ can be programmed with at most k operations under h-hot addressing.

In this paper we examine array programming under stores alone and under both stores and restores. We study stores alone for two reasons. First, array programming under stores is used to establish that array programming under stores and restores is **NP**-hard (see Section 5.4). Second, a store operation under some technologies may require much more time than a restore [7]. If so, we may be restricted to using restores alone, which is equivalent to programming with stores alone.

3.3. *Most Problems Require Many Steps*

We use a counting argument to show that most $n \times n$ instances of the array programming problem under h-hot addressing require a large number of programming steps. When $h = 1$ and n is large, this number is close to $n/2$. (n steps always suffice when $h = 1$; program the array by rows.) As h increases, the minimum number of steps for most arrays approaches $n^2/\log n$.

Theorem 3.1. *For $0 < \varepsilon < 1$ and n large, a fraction of at least $1 - 2^{-\varepsilon n^2}$ of the 2^{n^2} $n \times n$ arrays require at least $(1 - \varepsilon)n^2/(2b + 1)$ steps to program where $b = n$ when $h = 1$, $b = \sqrt{2n}$ when $h = 2$, $b = (6n)^{1/3}$ when $h = 3$, and $b = \log_2 n$ when $h = (\log_2 n)/2$.*

Proof. Let $\nu_h^0(k)$ (let $\nu_h^1(k)$) be the number of $n \times n$ arrays that can be programmed in k steps under stores alone (both stores and restores). Because there are 2^{n^2} possible such arrays, if $\nu_h^j(k) \leq 2^{(1-\varepsilon)n^2}$, a fraction of at least $1 - 2^{-\varepsilon n^2}$ of the arrays will require more than k_0^j programming steps where k_0^j satisfies $\nu_h^j(k_0^j) = 2^{(1-\varepsilon)n^2}$.

The first step in a program must be a store. Subsequent steps are stores or restores, if the latter are allowed. Let Γ_h be the number of subsets of rows or columns of an $n \times n$ array that can be selected under h-hot addressing. Then at most Γ_h^2 different stores and restores can be specified on any one step. It follows that $\nu_h^j(k) = 2^{j(k-1)}\Gamma_h^{2k}$ because, when restores are allowed, each operation other than the first can be either a store or a restore. (Note: $j = 0$ when only stores are used.) Combining these observations we have that k_0^j satisfies

$$j(k_0^j - 1) + 2k_0^j \log_2 \Gamma_h = (1 - \varepsilon)n^2.$$

Since the value of k_0^j when $j = 1$ is smaller than when $j = 0$, we use it as the lower

bound. Thus $1 - 2^{-\varepsilon n^2}$ of the arrays require k_0^1 or more steps. Since

$$\Gamma_h = \binom{b}{h} + \binom{b}{h+1} + \cdots + \binom{b}{b} \le 2^b,$$

where $n = \binom{b}{h}$, if we replace Γ_h by 2^b the solution k_1 satisfies $k_1 \le k_0^1$ where

$$(k_1 - 1) + 2k_1 b = (1 - \varepsilon)n^2 \quad \text{or} \quad k_1 \ge \frac{(1 - \varepsilon)n^2}{2b + 1}.$$

In summary, a fraction of at least $1 - 2^{-\varepsilon n^2}$ of the $n \times n$ arrays requires at least $(1 - \varepsilon)n^2/(2b + 1)$ steps.

To complete the analysis, we relate h and b to $n = \binom{b}{h}$ when n is large. Then $b = n$ when $h = 1$, $b = \sqrt{2n}$ when $h = 2$, and $b = (6n)^{1/3}$ when $h = 3$. This implies that when b is large, almost all $n \times n$ arrays require at least $(1 - \varepsilon)n/2$ steps when $h = 1$, $(1 - \varepsilon)n^{1.5}/2.83$ steps when $h = 2$, and $(1 - \varepsilon)n^{5/3}/3.63$ steps when $h = 3$. When $h = b/2$, we use Stirling's approximation [10] for the factorial function $n!$, which is given below:

$$\sqrt{2\pi n}\, n^n e^{-n} e^{\varepsilon_2(n)} \le n! \le \sqrt{2\pi n}\, n^n e^{-n} e^{\varepsilon_1(n)}.$$

Here $\varepsilon_1(n) = 1/(12n)$ and $\varepsilon_2(n) = 1/(12n + 1)$. This yields

$$n = \binom{b}{h} = \alpha \frac{2^{bH(h/b)}}{\sqrt{2\pi b(h/b)(1 - h/b)}},$$

where $H(x) = -x \log_2 x - (1-x) \log_2 (1-x)$ is the entropy function and $0.97 \le \alpha \le 1$ when $b \ge 8$. When $h = b/2$, $n = \alpha 2^b/\sqrt{\pi b/2}$ or $b = \log_2 n$ and $h = (\log_2 n)/2$ when n is large. Thus, almost all $n \times n$ arrays require at least $(1 - \varepsilon)n^2/2 \log_2 n$ programming steps in this case. □

Although most arrays require many steps to program, we show that strongly structured arrays are not among the hardest to program. We believe that many arrays of interest have a strong structure that makes them easier to program.

3.4. *Structured Arrays*

To acquire experience in array programming and understand the savings obtained from the exploitation of structure, we now introduce several simple $n \times n$ binary arrays $A = [a_{i,j}]$ that are analyzed in later sections. They are the **diagonal** ($a_{i,j} = 1$ only when $i = j$), **half-full** ($a_{i,j} = 1$ when $j \le i$), **ρ-lower-full** ($a_{i,j} = 1$ when $j \le i - \rho$ and $-(n-1) \le \rho \le (n-1)$, see Figure 5(a)), **$\rho$-upper-full** ($a_{i,j} = 1$ when $j \ge i - \rho$ and $-(n-1) \le \rho \le (n-1)$), **bandwidth-$\beta$** ($a_{i,j} = 1$ when $|i - j| \le \beta$), and ***s*-sparse** ($a_{i,j} = 1$ for at most s elements in each row and column) arrays. A **ρ-lower(upper)-triangular array** is a ρ-lower(upper)-full array when $\rho \ge 0$.

It is easy to see how these simple structures could appear in stored data patterns such as images or could arise when a crossbar is used to implement connections between horizontal and vertical wires, for example, to allow a set of parallel wires to "turn the corner."

4. Programming Structured Arrays

We now develop programs for structured arrays under stores and restores beginning with programs under 1-hot addressing. First, however, we derive lower bounds on the number of steps to program these arrays when only stores are used.

4.1. *Programming Structured Arrays with Stores*

The above structured arrays require at least proportional to n operations when only stores are used to program them.

Lemma 4.1. *When only stores are allowed, programming of $n \times n$ diagonal and half-full arrays under 1-hot addressing requires at least n operations. The ρ-lower(upper)-full arrays and the bandwidth-β arrays require at least $n - |\rho|$ and $n - \beta$ operations, respectively. The s-sparse arrays require at least $\lceil n/s \rceil$ stores.*

Proof. Since no two 1s in the diagonal array fall into a common activation matrix, each entry must be inserted with a separate store operation. The same statement applies to the main diagonal of a half-full array, the top (bottom) diagonal of the ρ-lower(upper)-full array which has $(n - |\rho|)$ 1s, and the top diagonal of the bandwidth bounded array of bandwidth-β array which has $(n - \beta)$ 1s.

For s-sparse arrays if we maximize the number of 1s in an activation matrix, we minimize the number of such matrices. Since an activation matrix cannot have more than s rows and s columns, the largest possible activation matrix involves s rows. Thus, at least $\lceil n/s \rceil$ activation matrices are needed to cover the n rows in the array. □

4.2. *Programming Structured Arrays with Stores and Restores*

We now show that restores can dramatically reduce the number of operations required to program many common arrays. We give $O(\log n)$ time algorithms to program diagonal, half-full, and banded $n \times n$ arrays and derive lower bounds under stores and restores showing that these upper bounds are best possible within a factor of two. We also present efficient algorithms to program s-sparse arrays.

The h-hot addressing scheme introduces dependencies between nanowires that make programming of arrays difficult. We simplify the programming of such arrays by using the *nested blocks model*.

4.3. *Nested Blocks Model for h-Hot Addressing*

In the **nested blocks model** the b address wires are partitioned into h equal-sized disjoint sets each containing $\bar{b} = \lfloor b/h \rfloor$ integers, $\{A^{(1)}, A^{(2)}, \ldots, A^{(h)}\}$. The ith component a_i

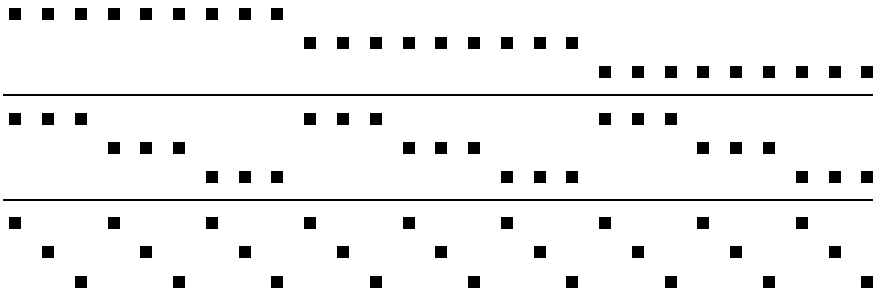


Fig. 2. Illustration of the three components of a 3-hot nested blocks encoding.

of an address $(a_1, a_2, \ldots, a_h)$ is chosen from $A^{(i)}$. Under this model $n = \bar{b}^h \leq (b/h)^h$ nanowires can be addressed, which is smaller than $\binom{b}{h}$ but still large, and $\bar{b} = n^{1/h}$.

This model naturally divides the column (row) nanowires into groups associated with the value of $a_1 \in A^{(1)}$. Within the group associated with a particular value for a_1 are $\bar{b}$ groups associated with the value of a_2, etc., as is illustrated in Figure 2 for $h = 3$ and $\bar{b} = 27$. In this model the $\bar{b} = n^{1/h}$ outer groups (determined by a_1) each contain $n_1 = \bar{b}^{h-1} = n^{1-1/h}$ nanowires. These nanowires are further subdivided into $\bar{b}$ groups each containing $n_2 = \bar{b}^{h-2} = n^{1-2/h}$ nanowires. In general, the first j components of an h-tuple specify a group containing $n_j = \bar{b}^{h-j} = n^{1-j/h}$ nanowires, $0 \leq j \leq h$. Note that $n_0 = n$.

To activate the group of $n_j = n^{1-j/h}$ column (row) nanowires defined by fixing the first j values of a nanowire address h-tuple, the address wires corresponding to the values of the first j components are activated along with all the address wires that correspond to all values of the last $h - j$ components of a nanowire address. Note that it is not possible to activate a contiguous set of nanowires without also activating other wires.

A block of an $n \times n$ array, $n = \bar{b}^h$, may be specified by fixing the first r components of a row address h-tuple and the first c components of a column address h-tuple. Such a specification identifies an $n_r \times n_c$ subarray of the $n \times n$ array.

When $h = 1$, this model imposes no restrictions on the addressing of nanowires.

4.4. *Diagonal Arrays under* 1*-Hot Addressing*

As shown in Lemma 4.1, n stores are required to program the $n \times n$ diagonal when only stores are allowed. We now show that this can reduced to $O(\log n)$ when restores are also allowed. This result obviously applies to any array with a single 1 in each row and column.

Theorem 4.1. *An algorithm exists to program the $n \times n$ diagonal array under* 1*-hot addressing using $2\lceil \log_2 n \rceil$ stores and restores.*

Proof. Below is a sketch of a recursive procedure $\mathcal{D}(n)$ to program the $n \times n$ diagonal array D in $T(n)$ steps when $n = 2^p$. (See Figure 3(b).)

1. In parallel execute $\mathcal{D}(n/2)$ in each quadrant of D in $T(n/2)$ steps.
2. Restore the lower left $n/2 \times n/2$ quadrant in one step.
3. Restore the upper left $n/2 \times n/2$ quadrant in one step.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

(a) (b)

Fig. 3. Arrays involved in the programming the diagonal array.

The parallel execution of $\mathcal{D}(n/2)$ in all four quadrants is done as follows: Assume that $\mathcal{D}(n/2)$ consists of a sequence of k stores and restores defined by the matrices $E^1_{R_1,C_1}$, $E^2_{R_2,C_2}, \ldots, E^k_{R_k,C_k}$ where each $E^j_{R_j,C_j}$ denotes either an activation or deactivation matrix. When $U \subseteq \{1, 2, \ldots, n/2\}$, let $(U + n/2)$ denote the set of integers obtained by adding $n/2$ to every entry in U. Let $R^*_j = R_j \cup (R_j + n/2)$ and $C^*_j = C_j \cup (C_j + n/2)$. Then parallel execution of $\mathcal{D}(n/2)$ in each quadrant of D means we apply the same sequence of stores and restores used by $\mathcal{D}(n/2)$ to (R^*_1, C^*_1), (R^*_2, C^*_2), …, (R^*_k, C^*_k), that is, we execute $E^1_{R^*_1,C^*_1}$, $E^2_{R^*_2,C^*_2}, \ldots, E^k_{R^*_k,C^*_k}$. These operations program a diagonal subarray into each quadrant of D.

The running time of the above procedure satisfies the recurrence $T(n) = T(n/2)+2$. Since the 2×2 array can be programmed in two steps, $T(2) = 2$. (See Figure 3(a).) It follows that $T(n) = 2\log_2 n$. When n is not a power of two, we can program the array as if it were a $2^m \times 2^m$ array, $m = \lceil \log_2 n \rceil$ and discard operations on non-existing rows and columns. Simple but careful analysis shows that this algorithm uses two stores and $2(\log_2 n - 1)$ restores. □

A slightly less efficient algorithm programs the $n \times n$ diagonal array using one store to fill the array with 1s and restores all off-diagonal positions using $2\lceil \log_2 n \rceil$ restores. We use this fact when programming other arrays.

Corollary 4.1. *An algorithm exists to program the $n \times n$ diagonal array in which all diagonal elements are entered in one store step and off-diagonal elements are restored in $2\lceil \log_2 n \rceil$ restore steps.*

4.5. *Diagonal Arrays under h-Hot Addressing*

We show that it is possible to program a diagonal array in $2\log_2 n + 2h$ steps, which is at most $2h + 1$ more steps than is used for 1-hot array programming.

Theorem 4.2. *The $n \times n$ diagonal array can be programmed using $2h\lceil (1/h)\log_2 n \rceil$ operations when h-hot addressing is used on b address wires where $n = (\lfloor b/h \rfloor)^h$.*

Proof. Our algorithm for h-hot addressing is recursive and based primarily on the algorithm of Corollary 4.1. This algorithm programs a diagonal $n \times n$ array using 1-hot addressing with one store operation and $2\lceil \log_2 \bar{b} \rceil$ restores. Let T_h be the number of operations to program a diagonal $\bar{b}^h \times \bar{b}^h$ array using this algorithm. For our base case, namely, to program diagonal a $\bar{b} \times \bar{b}$ diagonal array, we use the algorithm of Theorem 4.1. It uses $T_1 = 2\lceil \log_2 \bar{b} \rceil$ operations.

The algorithms of Theorem 4.1 and Corollary 4.1 treat a $\bar{b} \times \bar{b}$ array as if it had 2^t rows and columns, $t = \lceil \log_2 \bar{b} \rceil$, but ignores operations on rows and columns that do not exist.

To program a complete diagonal $\bar{b}^h \times \bar{b}^h$ array we view it as a $\bar{b} \times \bar{b}$ array whose entries are $\bar{b}^{h-1} \times \bar{b}^{h-1}$ arrays. We invoke the algorithm of Corollary 4.1. In T_{h-1} steps (replacing the one store) the algorithm programs a diagonal array into each $\bar{b}^{h-1} \times \bar{b}^{h-1}$ subarray. It then executes $2\lceil \log_2 \bar{b} \rceil$ restore operations in which individual elements in the algorithm of Corollary 4.1 are replaced by $\bar{b}^{h-1} \times \bar{b}^{h-1}$ subarrays. It follows that $T_h = T_{h-1} + 2\lceil \log_2 \bar{b} \rceil$ or $T_h = 2h\lceil (1/h) \log_2 n \rceil$. □

4.6. *Diagonal Arrays with Missing Elements*

Clearly, the upper bound of Theorem 4.1 can be generalized to diagonal arrays when not all diagonal elements are 1. If such an array has m 1s, it can be programmed under 1-hot in at most $2\lceil \log_2 m \rceil$ store and restore steps. It is much harder to program a diagonal array with stores and restores under h-hot when the array has missing 1s, as shown below.

Theorem 4.3. *The $n \times n$ diagonal array with missing elements can be programmed with $n^{1-1/h} + 2h\lceil (1/h) \log_2 n \rceil$ store and restore operations.*

Proof. We use the recursive algorithm of Theorem 4.2 to produce a diagonal array. Then we restore the missing 1s. There are $n^{1-1/h}$ innermost $n^{1/h} \times n^{1/h}$ blocks on the diagonal. In one step per block we restore all diagonal elements that are missing along with the entries in the subarray defined by these elements. Since the off-diagonal entries in each block that are restored are required to be 0, the desired result is produced. □

We establish a connection between the 2-hot version of this problem and the 1-hot array programming of an $n^{1/2} \times n^{1/2}$ array with stores and restores thereby demonstrating that if the latter problem can be solved quickly, then with an additional $O(\log n)$ restores we can quickly program the diagonal array with missing elements.

Theorem 4.4. *In the 2-hot addressing model a general $n \times n$ diagonal array with missing diagonal 1s can be programmed using at most as many stores and restores as are needed to program a related $\sqrt{n} \times \sqrt{n}$ array using 1-hot addressing followed by $2\lceil \frac{1}{2} \log_2 n \rceil$ restores.*

Proof. Consider the algorithm of Theorem 4.2 when $h = 2$. It operates on $n \times n$ arrays that are viewed as $\sqrt{n} \times \sqrt{n}$ arrays whose entries are $\sqrt{n} \times \sqrt{n}$ subarrays. Let C_i, $1 \le i \le \sqrt{n}$, denote the ith diagonal subarray and observe that the diagonal 1s in each

subarray can be specified by giving just their column positions. If several of the diagonal subarrays have 1s in the same column, they can be entered together in one step. The overlap between columns in different diagonal subarrays is made explicit in a $\sqrt{n} \times \sqrt{n}$ array in which the ith row has a 1 in the jth position if the jth column of C_i has 1 on its diagonal.

If we can program this array with a sequence of stores and restores using 1-hot addressing, the same sequence can be used to program the full array using 2-hot addressing. Off-diagonal 1s are restored in $2\lceil \frac{1}{2} \log_2 n \rceil$ operations following Corollary 4.1. □

4.7. *Lower Bound on the Complexity of Diagonal Array Programs*

We now show that the upper bound on the number of operations to program the full diagonal array is tight within a factor of two.

Theorem 4.5. *At least* $\lceil \log_2 n \rceil + 1$ *operations are required to program the* $n \times n$ *diagonal array under* 1*-hot addressing using stores and restores.*

Proof. It suffices to consider the $n \times n$ diagonal array that has n 1s in every diagonal position. Let q be the number of store operations that are used by the algorithm.

The algorithm may insert a 1 in a given diagonal position multiple times. For each i there is a last time when the ith diagonal element is inserted by the algorithm. Let m_j be the number of diagonal elements that are inserted for the last time by the jth store operation, $1 \le j \le q$. Since $\sum_{j=1}^{q} m_j \ge n$, it follows that there exists j_0 such that $m_{j_0} \ge n/q$.

The j_0th store operation not only inserts m_{j_0} diagonal elements, it fills an $m_{j_0} \times m_{j_0}$ subarray with 1s, $m_{j_0} \ge n/q$, as suggested by the bold 1s in Figure 4. Subsequent operations must contain sufficiently many restore operations to delete all the off-diagonal 1s in this subarray. The total number of restore operations to program the $n \times n$ diagonal array must be at least as large as the number to eliminate these 1s from this subarray.

Given an $m \times m$ array M of 1s, we derive a lower bound on $D(m)$, the number of restores needed to delete the off-diagonal 1s.

Let the first restore be on an $r \times c$ subarray N. Since N does not include any diagonal elements, it contains at most $m - r$ columns because if it contained more columns, these columns would contain diagonal elements of M. Thus, $r + c \le m$. Without loss of generality, let $r \le c$. Consequently, $r \le m/2$.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ 1 & 0 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Fig. 4. A subarray of 1s on which off-diagonal restores are needed.

Remove from M the r rows of N and the r columns containing the diagonals in these r rows, the resultant $(m-r)\times(m-r)$ array is full of 1s and has on its diagonal $m-r$ of the original diagonal elements. Since the off-diagonal 1s in this array are among those that must be restored, it follows that $D(m)\geq 1+D(m-r)$. Since $D(m)$ is a monotonically non-decreasing function of m and $r\leq m/2$, $D(m)\geq 1+D(\lceil m/2\rceil)$. Since $D(2)=1$, this implies that $D(m)\geq\lceil\log_2 m\rceil$.

It follows that $L(n)$, the total number of operations to program the $n\times n$ diagonal array, satisfies $L(n)\geq q+\lceil\log_2 n/q\rceil$. The right-hand side is a non-decreasing function whose minimum, $1+\lceil\log_2 n\rceil$, occurs at $q=1$ and $q=2$.

Clearly, the lower bound on the number of steps to program arrays under 1-hot applies to h-hot array programming as well. □

4.8. *Lower-Full and Upper-Full Arrays under* 1*-Hot Addressing*

To derive an upper bound on the number of steps to program a lower-full array under 1-hot, instead of restoring both the lower-left and upper-right quadrants of the array, as in the proof of Theorem 4.1, store 1s in the lower-left quadrant and restore 0s in the upper-right quadrant. To derive a lower bound, note that in the above proof only the 1s in the full $(n/q)\times(n/q)$ subarray above the diagonal need to be removed using only restores. Again, a lower bound of the form $D(m)\geq 1+D(\lceil m/2\rceil)$ exists on the number of restores to eliminate 1s above the main diagonal. We summarize these results below.

Theorem 4.6. *The $n\times n$ half-full array can be programmed using $2\lceil\log_2 n\rceil$ store and restore operations under 1-hot addressing. At least $\lceil\log_2 n\rceil+1$ operations are needed to program any such $n\times n$ array for any value of n.*

To program a ρ-upper-full array, use the algorithm for a ρ-lower-full array with the order of rows and columns inverted. To program an $n\times n$ ρ-lower-full array when $\rho>0$, observe that it constitutes an $(n-\rho)\times(n-\rho)$ half-full array when the first ρ rows and the last ρ columns of the array are deleted. To program an $n\times n$ ρ-lower-full array when $\rho<0$, observe that it is the complement of a $(|\rho|+1)$-upper-full array. Thus, it can be programmed by executing a store to fill the array with 1s followed by the operations to program a $(|\rho|+1)$-lower-full array in which stores and restores are exchanged and the row and column orders are reversed. A lower bound on the number of steps to program ρ-upper(lower)-full arrays is implied by Theorem 4.6.

Theorem 4.7. *The $n\times n$ ρ-lower(upper)-full array can be programmed using at most $2\lceil\log_2(n-|\rho|)\rceil+1$ store and restore operations. At least $\lceil\log_2(n-|\rho|)\rceil+1$ operations are needed to program any such $n\times n$ array for any value of n.*

4.9. *Lower-Full and Upper-Full Arrays under h-Hot Addressing*

We now describe highly efficient algorithms to program lower-full and upper-full structured arrays under h-hot addressing when both stores and restores are available.

The upper bound for h-hot programming of the diagonal array obtained in Theorem 4.2 is easily extended to the half-full array. Instead of restoring the lower left quadrant, 1s are stored in it. The same number of steps is used.

Theorem 4.8. *The $n \times n$ half-full array under h-hot addressing can be programmed in $2h\lceil(1/h)\log_2 n\rceil$ operations.*

We now derive an upper bound on the number of steps to program an $n \times n$ ρ-lower-full array. This bound grows exponentially in h.

Theorem 4.9. *The $n \times n$ ρ-lower-full array under h-hot addressing can be programmed in at most $2 \cdot 3^h\lceil(1/h)\log_2 n\rceil$ operations.*

Proof. We develop a $T(h, \rho)$-step algorithm A for ρ-lower-full arrays when $\rho \geq 0$ (the 1s fall below the main diagonal) and $n = \bar{b}^h$. To program such arrays when $\rho \leq 0$ (the 0s fall above the main diagonal), fill the array with 1s and then write 0s in a triangle in the upper right corner of the array by applying a variant of algorithm A that uses the same number of steps. Thus, $T(h, -\rho) \leq T(h, \rho) + 1$ when $\rho \geq 0$.

Represent a ρ-lower-full array M, $\rho \geq 0$, as a $\bar{b} \times \bar{b}$ array M_B of $n_1 \times n_1$ subarrays. (An example of a 2-lower full array is shown Figure 5(a).) Let $\sigma_1 = (\rho \bmod n_1) \geq 0$ and $\sigma_2 = (n_1 - \rho \bmod n_1) \geq 0$. The subarrays of M_B are of four types (see Figure 5(b)): (a) a full array F of 1s, (b) an empty array of 0s, (c) a σ_1-lower-full array (As and Bs), and (d) a $(-\sigma_2)$-lower-full array ($\underline{D}$s).

Let $q = \lceil(n-\rho)/n_1\rceil$. The first column of M_B contains q non-zero subarrays. Thus, the number of instances of $\underline{D}$ is $q - 1$. The number of instances of A and B is $\lceil q/2\rceil$ and $\lceil(q-1)/2\rceil$, respectively.

Let $M(m, h, \sigma)$ be an $m \times m$ array of $n_1 \times n_1$ subarrays that has the σ-lower-full $n_1 \times n_1$ array in each diagonal position, full arrays below it, and empty arrays above it

$$\left[\begin{array}{ccc|ccc|ccc|ccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0
\end{array}\right]$$

(a)

$$\left[\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\underline{D} & B & 0 & 0 & 0 & 0 & 0 & 0 \\
F & \underline{D} & A & 0 & 0 & 0 & 0 & 0 \\
F & F & \underline{D} & B & 0 & 0 & 0 & 0 \\
F & F & F & \underline{D} & A & 0 & 0 & 0 \\
F & F & F & F & \underline{D} & B & 0 & 0 \\
F & F & F & F & F & \underline{D} & A & 0
\end{array}\right]$$

(b)

Fig. 5. (a) A 2-lower-full 12×12 array blocked into 3×3 subarrays showing the four types of subarray that results. (b) A block $\bar{b} \times \bar{b}$ array of $n_1 \times n_1$ subarrays containing full subarrays (Fs), a diagonal of $-(n_1 - \rho \bmod n_1)$-lower-full ($\underline{D}$s) and a diagonal of $(\rho \bmod n_1)$-lower-full arrays (As and Bs). The three subarrays that have As, Bs, and $\underline{D}$s on their main diagonal all intersect either in full or empty blocks.

and which is programmed using h-hot addressing. Observe that instances of the $\underline{D}$s, the Fs below them, and 0 subarrays above them form an $M(q-1, h, -\sigma_2)$ subarray of M. Similarly, the blocks of M at the intersection of the rows and columns containing the As (Bs) form an $M(\lceil q/2\rceil, h, \sigma_1)$ ($M(\lceil (q-1)/2\rceil, h, \sigma_1)$) subarray.

To program M we program in sequence $M(q-1, h, -\sigma_2)$, $M(\lceil q/2\rceil, h, \sigma_1)$, and $M(\lceil (q-1)/2\rceil, h, \sigma_1)$. Because the last two subarrays intersect only on blocks where $M(q-1, h, -\sigma_2)$ is either full or empty, M will be programmed correctly.

Consider the recursive algorithm for $M(2^k, h, \sigma)$ that programs $M(2^{k-1}, h, \sigma)$ into each of the four $2^{k-1} \times 2^{k-1}$ quadrants of M_B, stores 1s into the lower left quadrant, and restores entries in the upper right quadrant. This algorithm eventually requires the programming of $M(1, h-1, \sigma)$, a single σ-lower-full $n_1 \times n_1$ array addressed with $(h-1)$-hot addressing. This can be done in $T(h-1, \sigma)$ steps. Let $T_L(2^k, h, \sigma)$ be the running time of this algorithm. It follows that $T_L(2^k, h, \sigma) = T_L(2^{k-1}, h, \sigma) + 2 = 2k + T(h-1, \sigma)$. The three matrices that are programmed to program M_B each have at most $q-1$ rows of $n_1 \times n_1$ blocks. Thus, in each case it suffices to let $k = \lceil \log_2 q \rceil$.

Using the observation that $T(h, -\rho) \le T(h, \rho) + 1$ it follows that

$$T(h, \rho) \le 6\lceil \log_2 q \rceil + 2T(h-1, \sigma_1) + T(h-1, \sigma_2) + 1.$$

Instead of computing the running time exactly, we compute $\overline{T}(h) = \max_\rho T(h, \rho)$. Using the fact that $q \le \bar{b}$, we have the following bound on $\overline{T}(h)$:

$$\overline{T}(h) \le 6\lceil \log_2 \bar{b} \rceil + 3\overline{T}(h-1) + 1 \le 3^{h-1}\overline{T}(1) + (6\lceil \log_2 \bar{b} \rceil + 1)(3^{h-1} - 1)/2.$$

When $h = 1$, the arrays are $\bar{b} \times \bar{b}$. $\overline{T}(1)$ is the number of steps to program ρ-lower-full array maximized over ρ when 1-hot addressing is used. From Theorem 4.7, this number is at most $2\lceil \log_2 \bar{b} \rceil + 1$ since the innermost blocks are $\bar{b} \times \bar{b}$ arrays. It follows that $\overline{T}(h) \le 3^h(2\lceil \log_2 \bar{b} \rceil)$. Finally, we note that $\bar{b} = n^{1/h}$. □

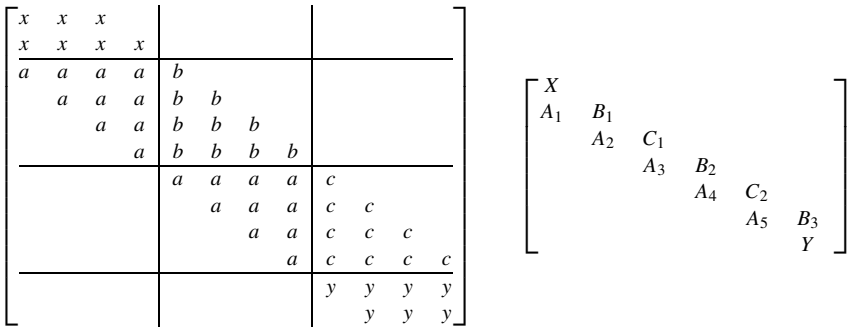
Since a ρ-lower-full $n \times n$ array under 1-hot addressing can be programmed in $2\lceil \log_2 n \rceil$ steps, it is clearly more difficult to program such arrays under h-hot addressing unless h is small. This argues for limiting the number of hot regions on nanowires.

4.10. *Banded Arrays under* 1*-Hot Addressing*

Similar results can be obtained for banded arrays of bandwidth β. We generalize the results to the case in which some band entries are 0.

Theorem 4.10. *The $n \times n$ banded array with bandwidth β, in which all band entries are* 1 *and 2β divides n, can be programmed using at most $6(\log_2 n + 2)$ store and restore operations. If some band entries are* 0*, at most 6β additional operations suffice to program the array.*

Proof. Let M be an $n \times n$ array of bandwidth β and let 2β divide n. The band elements (not all of which need to be 1) occupy the first $\beta + 1$ positions of the first row, the first $\beta + 2$ positions in the second, and $2\beta + 1$ in the $(\beta + 1)$st and later rows except for the



(a) (b)

Fig. 6. Decomposition of a banded array into subarrays.

last β rows where the number decreases by one until it reaches $\beta + 1$ in the nth row, as suggested in Figure 6(a) for an array of bandwidth $\beta = 2$.

The array M can be written as the block array shown in Figure 6(b) where X and Y are $\beta \times 2\beta$ arrays and A_i, B_i, and C_i are $2\beta \times 2\beta$ arrays. The subarrays $\{B_i\}$ and $\{C_i\}$ form the diagonals of non-intersecting block subarrays that can be programmed without affecting any of the subarrays in $\{A_i\}$.

The arrays $\{A_i\}$ are reflections about the diagonal of half-full arrays and the arrays $\{B_i\}$ and $\{C_i\}$ are half-full. Each can be programmed in $2\lceil \log_2 2\beta \rceil$ steps. The array X is the bottom half of a half-full array and Y is the top half of a reflected half-full array.

To program M we treat Y as a complete reflected half-full $2\beta \times 2\beta$ subarray and view it and the $2\beta \times 2\beta$ subarrays in $\{A_i\}$ as the diagonal of an $n/2\beta \times n/2\beta$ array M_A. We also treat X as a complete half-full $2\beta \times 2\beta$ subarray and view it and the $2\beta \times 2\beta$ subarrays in $\{C_i\}$ as the diagonal of an $n_c \times n_c$ array M_C where $n_c < n/2\beta$. Similarly, we view the $\{B_i\}$ as the diagonal of an $n_b \times n_b$ array M_B, $n_b < n/2\beta$.

When the band is full of 1s we program M_A using the algorithm of Theorem 4.6 on all the diagonal blocks in parallel. This programs M_A in $2\lceil \log_2 2\beta \rceil$ steps but leaves non-zero entries outside of the block diagonal. We then invoke the algorithm of Corollary 4.1 which uses $2\lceil \log_2 n/2\beta \rceil$ restores (array elements in this algorithm are replaced by $2\beta \times 2\beta$ subarrays) to restore all off-diagonal elements. Thus, a total of $2\lceil \log_2 2\beta \rceil + 2\lceil \log_2 n/2\beta \rceil$ steps suffice. We then program M_B and M_C in the same fashion. Since there is no overlap between them and they do not intersect with Y or the $\{A_i\}$, M can be programmed in $6\lceil \log_2 2\beta \rceil + 6\lceil \log_2 n/2\beta \rceil$ steps.

When the entries in the band are not all 0, we replace the stores in the algorithm of Theorem 4.6. For each of M_A, M_B, and M_C we program their $2\beta \times 2\beta$ diagonal arrays not in two steps but row by row in 2β parallel steps. Thus, at most 6β additional steps are necessary in this case. □

4.11. *Banded Arrays under h-Hot Addressing*

We now derive a bound on the number of steps to program a banded array when h-hot addressing is used. This bound is exponential in h.

$$\begin{bmatrix}
F & F & \underline{D_1} & 0 & 0 & 0 & 0 & 0 \\
F & F & F & \underline{D_2} & 0 & 0 & 0 & 0 \\
\overline{D_1} & F & F & F & \underline{D_3} & 0 & 0 & 0 \\
0 & \overline{D_2} & F & F & F & \underline{D_4} & 0 & 0 \\
0 & 0 & \overline{D_3} & F & F & F & \underline{D_1} & 0 \\
0 & 0 & 0 & \overline{D_4} & F & F & F & \underline{D_2} \\
0 & 0 & 0 & 0 & \overline{D_5} & F & F & F \\
0 & 0 & 0 & 0 & 0 & \overline{D_1} & F & F
\end{bmatrix}$$

Fig. 7. Banded array with $q_0 = 1$ and either $(\beta \bmod n_1) = 0$ or $(\beta \bmod n_1) = n_1 - 1$.

Theorem 4.11. *The $n \times n$ banded array of bandwidth β in which all band entries are* 1 *can be programmed using $O\left(3^h \max(\beta/n^{1-1/h}, 1)(1/h)\log_2 n\right)$ operations under h-hot addressing.*

Proof. Recall that $n_j = \bar{b}^{h-j}$ for $0 \le j \le h-1$. Thus, $n_0 = n$. We present a $T(h, \beta)$-step algorithm to program a banded array M of bandwidth β using h-hot addressing. Let $q_j = \lfloor \beta/n_{j+1} \rfloor - 1$ for $0 \le j \le h-1$. q_j is a non-decreasing function of j and $q_0 = \lfloor \beta/n_1 \rfloor - 1$. Represent M as a $\bar{b} \times \bar{b}$ array M_B containing $n_1 \times n_1$ subarrays.

Consider five different cases for β, namely, (I) $\beta = 0$, (II) $(\beta \bmod n_1) = 0$ but $\beta \neq 0$, (III) $(\beta \bmod n_1) = n_1 - 1$, (IV) $0 < (\beta \bmod n_1) < n_1 - 1$ and $q_0 \ge 0$, and (V) $0 < (\beta \bmod n_1) < n_1 - 1$ and $q_0 = -1$.

Case I is the diagonal array under h-hot addressing for which an algorithm has been given as well as an upper bound derived of $2h\lceil(1/h)\log_2 n\rceil$ (see Theorem 4.2).

Case II (see Figure 7) contains a block banded array of bandwidth q_0 that has full arrays (F) within the band and a diagonal of half-full arrays ($\underline{D_i}$) above the band and a diagonal of upper half-full arrays ($\overline{D_i}$) below the band. Each of these diagonals has $d = \bar{b} - (q_0 + 1)$ blocks. (The numbering of these arrays is explained below.) The following algorithm will program the array M_B in this case:

1. Program the banded array of Fs in $6(\log_2 \bar{b} + 2)$ steps (see Theorem 4.10.)
2. Organize blocks on the upper diagonal into $2q_0 + 2$ groups separated by $2q_0 + 1$ rows and columns. (In Figure 7 the arrays in the ith group are labeled $\underline{D_i}$.) The ith subarray containing instances of half-full $\underline{D_i}$ on its diagonal, $1 \le i \le 2q_0+2$, does not intersect another such subarray nor the band of Fs. This subarray has at most $m = \lceil d/(2q_0 + 2)\rceil \le \bar{b}$ rows and columns.

   Program the ith $m \times m$ block subarray using $(h-1)$-hot addressing by programming the $n_1 \times n_1$ half-full array $\underline{D_i}$ into every block in $2(h-1)\lceil(1/(h-1))\log_2 n_1\rceil = 2(h-1)\lceil\log_2 \bar{b}\rceil$ steps (see Theorem 4.8) and then restore all blocks outside the diagonal in $2\lceil\log_2 m\rceil \le 2\lceil\log_2 \bar{b}\rceil$ steps (see Corollary 4.1). The number of steps is bounded by $2h\lceil\log_2 \bar{b}\rceil$. All $(2q_0+2)$ groups of subarrays can be programmed in $2(2q_0 + 2)h\lceil\log_2 \bar{b}\rceil$ steps.
3. Program the $\overline{D}$s in $2q_0 + 3$ groups. (In this case the lower diagonal blocks are organized into groups that do not intersect any of the other blocks except for common empty blocks.)

$$\begin{bmatrix}
F & F & \underline{A_1} & \underline{B_1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
F & F & \overline{F} & \underline{A_2} & \underline{B_2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\overline{A_1} & F & F & \overline{F} & \underline{A_3} & \underline{B_3} & 0 & 0 & 0 & 0 & 0 & 0 \\
\overline{B_1} & \overline{A_2} & F & F & \overline{F} & \underline{A_4} & \underline{B_4} & 0 & 0 & 0 & 0 & 0 \\
0 & \overline{B_2} & \overline{A_3} & F & F & \overline{F} & \underline{A_1} & \underline{B_5} & 0 & 0 & 0 & 0 \\
0 & 0 & \overline{B_3} & \overline{A_4} & F & F & \overline{F} & \underline{A_2} & \underline{B_6} & 0 & 0 & 0 \\
0 & 0 & 0 & \overline{B_4} & \overline{A_5} & F & F & \overline{F} & \underline{A_3} & \underline{B_1} & 0 & 0 \\
0 & 0 & 0 & 0 & \overline{B_5} & \overline{A_1} & F & F & \overline{F} & \underline{A_4} & \underline{B_2} & 0 \\
0 & 0 & 0 & 0 & 0 & \overline{B_6} & \overline{A_2} & F & F & \overline{F} & \underline{A_1} & \underline{B_3} \\
0 & 0 & 0 & 0 & 0 & 0 & \overline{B_7} & \overline{A_3} & F & F & \overline{F} & \underline{A_2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \overline{B_1} & \overline{A_4} & F & F & \overline{F} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \overline{B_2} & \overline{A_5} & F & F
\end{bmatrix}$$

Fig. 8. Banded array satisfying $0 < (\beta \bmod n_1) < n_1 - 1$ and $q_0 = 1$.

The bound on the number of steps used here is obtained from that of the previous step in which we replace $2q_0 + 2$ by $2q_0 + 3$ or $2(2q_0 + 3)h\lceil \log_2 \bar{b} \rceil$ steps.

Thus, $6(\log_2 \bar{b} + 2) + 2(4q_0 + 5)h\lceil \log_2 \bar{b} \rceil$ steps suffice to program Case II.

Case III, illustrated in Figure 7 for $q_0 = 1$, applies when $(\beta \bmod n_1) = n_1 - 1$. This array is essentially the same as the one in Figure 7 except that the $\underline{D_i}$ and $\overline{D_i}$ arrays are not half-full arrays. $\underline{D_i}$ is 1-lower-full and $\overline{D_i}$ is 1-upper-full.

The time bounds given above apply after replacing the time to program a half-full array by the time to program a ρ-lower-full array. That is, we replace $2(h-1)\lceil \log_2 \bar{b} \rceil$ by $2 \cdot 3^{h-1}\lceil \log_2 \bar{b} \rceil$. Thus, the total number of steps to program the array in this case is at most $6(\log_2 \bar{b} + 2) + (4q_0 + 5)\left[2 \cdot 3^{h-1}\lceil \log_2 \bar{b} \rceil + 2\lceil \log_2 \bar{b} \rceil\right]$.

Case IV, which holds when $0 < (\beta \bmod n_1) < n_1 - 1$ and $q_0 \geq 0$, is illustrated in Figure 8. The band and the diagonals immediately above and below can be programmed exactly as in Case III in the same number of steps. The top and bottom diagonals are programmed the same way as diagonals in Case III except that the top diagonal is covered by $2q_0 + 4$ block subarrays and the bottom is covered by $2q_0 + 5$ block subarrays.

Thus, in this case, the array can be programmed in $6(\log_2 \bar{b} + 2) + (8q_0 + 14)[2 \cdot 3^{h-1}\lceil \log_2 \bar{b} \rceil + 2\lceil \log_2 \bar{b} \rceil]$. Thus, that the number of steps here is $O\left(3^h(q_0 + 1)\log_2 \bar{b}\right)$. Note that $q_0 + 1 \geq 1$ in this case.

Case V holds when $0 < (\beta \bmod n_1) < n_1 - 1$ and $q_0 = -1$. Here the main diagonal is composed of banded $n_1 \times n_1$ arrays of bandwidth β. The diagonals above and below the main diagonal can be programmed in the same manner as the corresponding diagonals in Case III with $q_0 = 0$.

Since $T(h, \beta)$ is the number of steps to program a banded $n_0 \times n_0$ array using h-hot addressing, Case V requires $T(h-1, \beta) + [10 \cdot 3^{h-1}\lceil \log_2 \bar{b} \rceil + 2\lceil \log_2 \bar{b} \rceil]$.

In summary, if Case V does not apply, $T(h, \beta)$ is no more than the largest of the bounds for the other four cases, which occurs in Case IV. Since q_j is a non-decreasing function, if $q_0 \geq 0$, Case V does not occur and $T(h, \beta) = O\left(3^h(q_0 + 1)\log_2 \bar{b}\right)$. If Case V does occur, then $T(h, \beta) = T(h-1, \beta) + O\left(3^h \log_2 \bar{b}\right)$. Since $T(1, \beta) \leq 6\log \bar{b} + 2$, this recurrence implies that $T(h, \beta) = O\left(3^h \log_2 \bar{b}\right)$. Thus, $T(h, \beta) = O\left(3^h \max(\beta/n^{1-1/h}, 1)(1/h)\log_2 n\right)$. □

4.12. *Sparse Arrays under* 1*-Hot Addressing*

An ***s*-sparse array** is an array that has at most s 1s in each row and column. We derive an upper bound on the number of operations to program such an array.

Theorem 4.12. *Every s-sparse $n \times n$ array, $n = \mu^k$ for $\mu = s(s-1)+1$, can be programmed with $O\left(s^2 \log^2(n/s^2)\right)$ operations.*

Proof. To program an arbitrary s-sparse $n \times n$ array, group the rows into sets such that within each set no two rows have 1s in common columns. The rows in each set form a subarray.

The first such subarray has at least $m_1 = \lceil n/\mu \rceil$ rows. This follows because (a) for each row in the subarray, there are at most $s(s-1)$ other rows that may overlap with it, $s-1$ rows for each of the s 1s in a row (each column contains at most s 1s), (b) to ensure no overlap between rows, $\mu = s(s-1)+1$ rows may have to be excluded for each row in the array, and (c) if $m_1\mu < n$, an additional non-overlapping row could be added.

After removing the first subarray, the array has $n - m_1$ remaining rows. The process can be repeated to identify a subarray with $m_2 = \lceil (n-m_1)/\mu \rceil$ rows. In general, a jth subarray with m_j rows can be identified where

$$m_j = \lceil \left(n - m_1 - m_2 - \cdots - m_{j-1}\right)/\mu \rceil.$$

When $n = \mu^k$, $m_j = (n/(\mu-1))((\mu-1)/\mu)^j$ for $j \geq 1$, as can be shown by induction. Thus $p = \max\{j \mid m_j \geq 1\} = \lfloor \log_2(n/(\mu-1))/\log_2(\mu/(\mu-1)) \rfloor$. We now bound the number of stores and restores to program one $m_j \times n$ subarray.

We permute the columns within each subarray so that the 1s in the first row are in the first s columns, those in the second row are in the next s columns, etc. If the jth row has less than s 1s, some entries in column $(j-1)s+1$ through column js may be 0. Let b_j be the $1 \times s$ block consisting of these entries in the jth row. To program this subarray, view it as a diagonal $m_j \times m_j$ array whose diagonal elements are $b_1, b_2, \ldots, b_{m_j}$. In parallel in one step program b_j into the jth set of columns for $j = 1, 2, \ldots, m_j$. Restore off-diagonal blocks in at most $2\lceil \log_2 m_j \rceil$ steps using the algorithm of Corollary 4.1. Thus, $2\lceil \log_2 m_j \rceil + 1 = 2\lceil \log_2 n/(\mu-1) - j \log_2 \mu/(\mu-1) \rceil + 1$ steps suffice to program the jth subarray.

Let $T(s,n)$ be the total number of operations to program the s-sparse $n \times n$ array when $n = \mu^k$. It follows that $T(s,n)$ has the following bound ($\lceil x \rceil \leq x+1$):

$$\begin{aligned} T(s,n) &\leq \sum_{j=1}^{p} 2\left(\log_2 n/(\mu-1) - j\log_2(\mu/(\mu-1)) + 2\right) \\ &= 2p\log_2 n/(\mu-1) - p(p-1)\log_2(\mu/(\mu-1)) + 2p. \end{aligned}$$

Here $p = \lfloor \log_2(n/(\mu-1))/\log_2(\mu/(\mu-1)) \rfloor$. Since $\ln(1+\varepsilon) \geq 0.8 \cdot \varepsilon$ for $\varepsilon \leq .5$, if we let $\varepsilon = 1/s(s-1)$, $1/\ln(\mu/(\mu-1)) \leq 1.25\, s(s-1)$ and we have the following inequality:

$$T(s,n) = O\left(\log_2^2(n/(\mu-1))/\log(\mu/(\mu-1))\right) = O\left(s^2 \log^2(n/(\mu-1))\right),$$

which is the desired conclusion. □

5. The Complexity of Array Programming

In Section 3.3 we show that most arrays require many steps whereas in Section 3.4 we show that some structured arrays can be programmed with a small number of steps. In this section we ask how difficult it is to identify the minimum number of steps to program arbitrary arrays. Below we show that this problem is **NP**-hard under stores and restores. As a consequence, good approximation algorithms are desirable. Unfortunately, as shown in Section 6, good approximations cannot be found under stores alone in polynomial time (assuming $\mathbf{P} \neq \mathbf{NP}$) unless special conditions exist on (a) the arrays being programmed, (b) the operations being performed, or (c) the size of h relative to b. The question of whether the approximation problem is hard under stores and restores remains open. These negative results lead us to examine in Section 8 heuristics for array programming that exploit structure.

5.1. *Continuous Reductions between Problems*

We employ Simon's [24] classification of optimization problems with respect to their approximability. Instances of maximization and minimization problems are defined by pairs (I, b) in which I has a value denoted $\nu(I)$. The "Yes" instances of such minimization (maximization) problems are those for which $\nu(I) \leq b$ ($\nu(I) \geq b$).

Consider a polynomial-time reduction $\rho : \mathcal{A} \mapsto \mathcal{B}$ between two maximization (or minimization) problems $\mathcal{A}$ and $\mathcal{B}$ that maps an instance (I, b) of $\mathcal{A}$ to an instance $(I', b') = \rho(I, b)$ of $\mathcal{B}$ such that the former is a "Yes" of $\mathcal{A}$ if and only if the latter is a "Yes" instance of $\mathcal{B}$.

Definition 5.1. A reduction $\rho : \mathcal{A} \mapsto \mathcal{B}$ between two maximization (or minimization) problems (I, b) and (I', b') is (a, c)-**bounded** if $\nu(I') \geq (\nu(I)/a - c)$ (or $\nu(I') \leq (\nu(I)/a - c)$). A reduction ρ is **asymptotically continuous** if there is an inverse reduction $\gamma : \mathcal{B} \mapsto \mathcal{A}$ such that ρ is (a, c)-bounded and γ is (d, e)-bounded for $a, d > 0$ and $c, e \geq 0$.

Note that if ρ is asymptotically continuous and $(I', b') = \rho(I, b)$, the values of $\nu(I)$ and $\nu(I')$ are linearly bounded by one another. Thus, a good approximation for one problem is a good one for the other.

We show that d-SET BASIS and COVERING BY COMPLETE BIPARTITE SUBGRAPHS, two **NP**-complete problems, are equivalent to ARRAY PROGRAMMING. The reductions given between these problems are asymptotically continuous. We use the reductions to show that ARRAY PROGRAMMING is **NP**-complete and, in Section 6, that it is **NP**-hard to find a close approximation to the related minimization problem.

d-SET BASIS
Instance: Triples $(U, \mathcal{S}, k)$ where $U = \{e_1, \ldots, e_m\}$ and $\mathcal{S}$ is a collection of sets, $\mathcal{S} = \{S_1, \ldots, S_n\}$, $S_i \subseteq U$, $|S_i| \leq d$ for $1 \leq i \leq n$, and k is an integer.
Answer: "Yes" if $(U, \mathcal{S}, k)$ has a **basis** of size $l \leq k$, that is, a collection of subsets $\mathcal{B} = \{B_1, \ldots, B_l\}$, $B_i \subseteq U$, such that for each i, $S_i \in \mathcal{S}$ is the union of some sets in $\mathcal{B}$.

SET BASIS is a generalized form of d-SET BASIS in which no limit is placed on the size d of basis sets. Stockmeyer [25] has shown that d-SET BASIS and SET BASIS are **NP**-complete.

A **biclique** is a complete bipartite subgraph. A set of bicliques **covers** a graph G if each biclique edge is in G and every edge in G is in some biclique. Clearly, some edges may be in multiple bicliques.

COVERING BY COMPLETE BIPARTITE SUBGRAPHS (CCB)
Instance: Pairs (G, k) where $G = (X \cup Y, E)$, $E \subseteq X \times Y$ is a bipartite graph and k is an integer.
Answer: "Yes" if there exists a set of at most k bicliques of G that covers all the edges of G.

5.2. *Reductions between* AP *under Stores Alone*, SET BASIS, *and* CCB

We now show that there are asymptotically continuous reductions between ARRAY PROGRAMMING under stores alone, SET BASIS, and CCB. Because AP is in **NP**, it is **NP**-complete.

Theorem 5.1. *There exist asymptotically continuous reductions between* AP *under stores alone*, SET BASIS, *and* CCB.

Proof. SET BASIS is equivalent to AP. Equate an instance $(U, \mathcal{S}, k)$ of AP, where $U = \{e_1, \ldots, e_m\}$ and $\mathcal{S} = \{S_1, \ldots, S_n\}$, with an instance $(\mathcal{W}, k)$ where $\mathcal{W}$ is an $n \times n$ array over $\{0, 1\}$ such that S_i corresponds to the ith row in $\mathcal{W}$, that is, $w_{i,j} = 1$ if $e_j \in S_i$. If $\{B_1, \ldots, B_l\}$ is a basis for $\mathcal{S}$, then, for $1 \leq j \leq l$, B_j defines an activation matrix containing row i if $B_j \subseteq S_i$ and contains the columns corresponding to entries in B_j. Similarly, given an activation matrix, a corresponding basis set exists. Thus, $(U, \mathcal{S}, k)$ is a "Yes" instance of SET BASIS if and only if $(\mathcal{W}, k)$ is a "Yes" instance of AP.

AP is also equivalent to CCB. Equate an instance of $((X \cup Y, E), k)$ of CCB with an instance $(\mathcal{W}, k)$ of AP where vertices in X correspond to rows in $\mathcal{W}$, vertices in Y correspond to columns in $\mathcal{W}$, and an edge exists between vertices in X and Y if and only if there is a 1 at the intersection of the corresponding row and column in $\mathcal{W}$. Since bicliques of $((X \cup Y, E), k)$ correspond to activation matrices of $\mathcal{W}$, $((X \cup Y, E), k)$ is a "Yes" instance of CCB if and only if $(\mathcal{W}, k)$ is a "Yes" instance of AP. □

Corollary 5.1. ARRAY PROGRAMMING *is* **NP**-*complete under stores alone*.

5.3. 1-*Hot Array Programming with* $O(\log n)$ *Steps*

A restricted version of ARRAY PROGRAMMING under stores, namely, when the number of operations is logarithmic in the size of an array, is also **NP**-complete, a result that is used below to show that ARRAY PROGRAMMING is **NP**-complete under both stores and restores. This result also implies that CCB is **NP**-complete when restricted to bipartite graphs that can be covered by a number of bicliques logarithmic in $|V|$, and SET BASIS is **NP**-complete when restricted to a collection of sets for which there is a basis containing a number of sets logarithmic in the cardinality of the universe.

ARRAY PROGRAMMING WITH $O(\log n)$ STORES (AP-log)
Instance: $(\mathcal{W}, k)$ where $\mathcal{W}$ is an $n \times n$ array over $\{0, 1\}$ and $k = O(\log n)$.
Answer: "Yes" if $\mathcal{W}$ can be programmed with at most k operations.

Theorem 5.2. AP-log *is* **NP**-*complete under stores alone*.

Proof. See the Appendix. □

5.4. *Array Programming under Stores and Restores*

We use the above fact and knowledge of a particular array for which the optimal algorithm under stores and restores uses only stores to show that under stores and restores ARRAY PROGRAMMING is **NP**-complete when "don't cares" are allowed. A don't care is an entry in an array $\mathcal{W}$ that is not specified; it may be programmed as either 0 or 1. Allowing don't cares in the array programming problem with only stores does not change its complexity; the problem remains **NP**-complete.

Our approach has two steps. First we show the existence of an $n \times n$ array $\mathcal{A}, n = 2^p$, that has a unique optimal program under stores and restores that uses only $\log_2 n$ stores. Second, we reduce in polynomial time the CCB-log decision problem under just stores to the problem of programming under stores and restores an array $\mathcal{Q}$ shown in Figure 9 that may contain don't cares.

Theorem 5.3. *There exists an $n \times n$ array $\mathcal{A}$, $n = 2^p$, for which the optimal program under stores and restores uses $\log_2 n$ stores and no restores.*

Proof. We prove the existence of an array $\mathcal{A}$ with the stated properties that has don't cares, although there exists another array with this property that doesn't have don't cares.

Let $\mathcal{B}$ have 1s on its diagonal, 0s above the diagonal, and don't cares below the diagonal. (See Figure 10(a).) Analysis identical to that of Theorem 4.5 except that $D(m)$ has 1s above its diagonal and don't cares below, which does not change the result, demonstrates that programming this array requires precisely $1 + \log_2 n$ store and restore steps. More specifically, the array may be programmed in one of only two ways: (a) one store that fills the array with 1s followed by $\log_2 n$ restores or (b) two stores that do not fill the entire array, followed by $\log_2 n - 1$ restores. If follows that if we were to start with an array filled with 1s instead of one filled with 0s, an optimal algorithm under stores and restores would use $\log_2 n$ restores to program the array.

Consider the array $\mathcal{A}$ that has 0s on the diagonal, 1s above the diagonal, and don't cares below the diagonal. (See Figure 10(b).) $\mathcal{A}$ is the complement of $\mathcal{B}$, that is, all but the don't care entries are complemented. It follows that exactly $\log_2 n$ stores are required to program array $\mathcal{A}$ under stores and restores. □

$$\mathcal{Q} = \begin{bmatrix} \mathcal{M} & \Phi \\ \Phi & \mathcal{A} \end{bmatrix}$$

Fig. 9. The $\mathcal{Q}$ array.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 0 & 0 \\ d & d & 1 & 0 & 0 \\ d & d & d & 1 & 0 \\ d & d & d & d & 1 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ d & 0 & 1 & 1 & 1 \\ d & d & 0 & 1 & 1 \\ d & d & d & 0 & 1 \\ d & d & d & d & 0 \end{bmatrix}$$

(a) (b)

Fig. 10. The $\mathcal{B}$ and $\mathcal{A}$ matrices. The d entries are don't cares.

Theorem 5.4. 1-*Hot array programming under stores and restores is* **NP**-*hard*.

Proof. We reduce AP-log under stores alone to AP under stores and restores. Let $\mathcal{M}$ be an $n \times n$ array that is an instance of AP-log. We create the array $\mathcal{Q}$ of Figure 9 that is composed of the array $\mathcal{M}$ in its upper left corner and the $m \times m$ array $\mathcal{A}$ described above, $m = 2^k$, in the lower right corner, and don't cares elsewhere.

Since array $\mathcal{A}$ requires exactly k store steps under stores and restores and array $\mathcal{A}$ is a subarray of array $\mathcal{Q}$, it follows that if array $\mathcal{Q}$ can be programmed in k steps, then all these steps must be store steps. Note that the don't care matrices allow any store step on subarray $\mathcal{A}$ to be programmed simultaneously with any store step on subarray $\mathcal{M}$ without affecting any components of array $\mathcal{Q}$. It follows that array $\mathcal{Q}$ can be programmed with k store and restore steps if and only if array $\mathcal{M}$ can be programmed with k stores.

It follows that AP under stores and restores can be used to solve AP-log under stores alone. It follows that AP under stores and restores is **NP**-hard. Since the number of rows and columns of $\mathcal{Q}$ is polynomial in the number of rows and columns of $\mathcal{M}$, the reduction can be done in polynomial time. □

Because it is **NP**-hard under stores and restores to find optimal solutions to the 1-hot $n \times n$ array programming problem, the same is true for h-hot array programming, $h > 1$.

Theorem 5.5. *h-Hot array programming under stores and restores is* **NP**-*hard*.

This result follows because an instance of a 1-hot problem can be embedded in an instance of an h-hot problem. This motivates the study of programming of individual rows of an array under h-hot addressing in Section 7.

6. Approximation Algorithms under Stores

The complexity of array programming and related problems naturally suggests that good approximation algorithms be sought. Unfortunately, such algorithms are not possible under stores alone unless **P = NP**. The question of whether such good approximation algorithms exists under stores and restores is open.

Simon [24] has shown that CCB is NP-complete by asymptotically continuous reduction from clique partition defined below:

CLIQUE PARTITION (CP)
Instance: Pairs (G, k) where $G = (V, E)$ is a graph and k is an integer.
Answer: "Yes" if there exists a set of at most k vertex-disjoint cliques in G such that every vertex of G is covered by some clique.

Since CLIQUE PARTITION is not approximable in polynomial time to within a multiplicative factor of $|V|^{\varepsilon}$ for some $\varepsilon > 0$ if $\mathbf{P} \neq \mathbf{NP}$ [19], the continuous reduction to CCB implies that the inapproximability of clique partition applies to CCB, as has been noted by Hochbaum [12], and also to AP and SET BASIS.

Corollary 6.1. CCB, AP, *and* SET BASIS *are not approximable to within* N^{ε} *of optimal for some* $\varepsilon > 0$ *in polynomial time if* $\mathbf{P} \neq \mathbf{NP}$, *where* N *is* $|V|$ *for* CCB, $n + m$ *for* AP, *and the number of sets plus the cardinality of the universe for* SET BASIS.

Because CCB, AP, and SET BASIS are very difficult to approximate using a deterministic algorithm if $\mathbf{P} \neq \mathbf{NP}$, one asks if good randomized algorithms might be found for them. This question is equivalent to the same question for CLIQUE PARTITION and GRAPH COLORING because of the asymptotically continuous reductions that exist between them and CCB [24]. It follows that if a good randomized algorithm exists for any of these five problems, a good randomized algorithm exists for all of them. In Section 8 we explore heuristics for array programming when both stores and restores are allowed.

6.1. *Bounded Arrays and Bounded Activation Matrices*

While there exist strong inapproximability results for array programming and covering by complete bipartite graphs, in this section we demonstrate that good bounds can be obtained when the number of 1s in each row and column of an array are bounded (the degree of vertices in bipartite graphs is bounded) or the numbers of rows or columns of the activation matrices are bounded. These bounds are **log-approximate**, that is, they are within a factor of $O(\log N)$ of the optimal bounds where N is the size of a problem description. For $n \times n$ arrays, $N = n^2$.

6.1.1. *The* d-AP *and* d-CCB *problems*. There is a possibility that technology may impose a limit on the number of 1s in each row or column of an array. Since 1s correspond to connections, too many 1s may result in a voltage drop that is excessive. Motivated by these considerations, we define the following problems.

Definition 6.1. d-AP consists of instances $(\mathcal{W}, k)$ of AP in which the number of 1s in each row (or column) of $\mathcal{W}$ is at most d. d-CCB is the version of CCB in which the degree of each $x_i \in X$ (or $y_j \in Y$) is less than or equal to d.

Theorem 6.1. *There exist asymptotically continuous reductions between* d-AP, d-SET BASIS *and* d-CCB.

Proof. The proof is identical to that of asymptotically continuous reductions between AP, SET BASIS, and CCB. □

6.1.2. *Approximations to* d-AP, d-SET BASIS, *and* d-CCB. Although there exist strong inapproximability results for CCB, AP, and SET BASIS, the problems d-AP, d-SET BASIS, and d-CCB admit log-approximation algorithms in polynomial time. This follows from

a recasting of d-AP as a version of SET COVER, a well-studied problem for which a log-approximation exists.

SET COVER
Instance: Triples $(U, \mathcal{S}, k)$ where $U = \{e_1, \ldots, e_m\}$ and $\mathcal{S}$ is a collection of sets $\mathcal{S} = \{S_1, \ldots, S_n\}$, $S_i \subseteq U$, and k is an integer.
Answer: "Yes" if there exists a collection $\mathcal{S}' \subseteq \mathcal{S}$, $|\mathcal{S}'| \leq k$, such that the union of all sets in $\mathcal{S}'$ is equal to U.

Karp [16] demonstrated that SET COVER is **NP**-complete. Johnson [15] has shown that it admits a log-approximation.

We now demonstrate that d-AP, d-SET BASIS, and d-CCB may each be modeled as instances of the SET COVER problem.

Theorem 6.2. *d-AP, d-SET BASIS, and d-CCB each admit a* log*-approximation algorithm under stores alone.*

Proof. Since there are asymptotically continuous reductions between all three problems, it suffices to demonstrate a log-approximation algorithm for d-AP. We cast an instance $(\mathcal{W}, k)$ of d-AP as an instance $(U, \mathcal{S}, k)$ of SET COVER. Here $\mathcal{W} = \left[w_{i,j}\right]$ is an $n \times n$ array over the set $\{0, 1\}$ with at most d 1s in each row (column). In the instance of set cover $U = \left\{(i, j) \mid w_{i,j} = 1\right\}$ and $\mathcal{S}$ is a collection of subsets of U corresponding to stores. Without loss of generality we assume that each store is maximal, that is, it involves all the rows having 1s in a given set of columns.

A "Yes" instance of $(\mathcal{W}, k)$ is one that can be programmed with at most k stores. A "Yes" instance of $(U, \mathcal{S}, k)$ is one for which there exists a set cover of U, a collection of at most k sets from $\mathcal{S}$ that covers all the elements of U. Clearly, there is a one-to-one correspondence between the "Yes" instances of the two problems. We now show that this reduction can be implemented in polynomial time.

The reduction is obtained by constructing U and $\mathcal{S}$ from $\mathcal{W}$. To construct U we enumerate all pairs of indices corresponding to non-zero elements of $\mathcal{W}$. To assemble the set of possible maximal stores we enumerate the sets of d or fewer columns of $\mathcal{W}$ and for each set find the rows of $\mathcal{W}$ that have 1s in each column in the set. It follows that n comparisons suffice to find the rows containing 1s in a given set of columns.

The number $\sigma(m, d)$ of subsets of U (m columns) containing d or fewer column indices satisfies

$$\sigma(m, d) = \binom{m}{d} + \binom{m}{d-1} + \cdots + \binom{m}{1}.$$

It is easy to demonstrate that $\sigma(m, d) = O(m^d)$, when d is a constant, which is polynomial in the size of an instance $(\mathcal{W}, k)$.

Combining these observations we conclude that we can construct an instance $(U, \mathcal{S}, k)$ of SET COVER in time polynomial in the size of an instance $(\mathcal{W}, k)$.

It follows that the well-known greedy algorithm [15] for SET COVER provides a log-approximation for d-AP, d-SET BASIS, and d-CCB. □

6.2. *Approximability of* AP *with Bounded Activation Matrices*

Alternatively, there is a possibility that technology may impose a limit on the number of 1s in each row or column of an activation matrix. Since 1s correspond to connections, it is more likely that a voltage drop will limit the size of activation matrices than it will limit the number of 1s in an array. The next result follows immediately from Theorem 6.2.

Theorem 6.3. *Array programming where the size of the activation matrices is bounded is* log*-approximable under stores alone.*

We next show that h-hot array programming becomes log-approximable when h and b, the number of address wires, are both logarithmic in n for $n \times n$ arrays.

6.3. *Log-Hot Programming is Log-Approximable*

While the problem of finding a near minimum number of steps to program an array under h-hot addressing is hard, the problem becomes log-approximable when h is large, as we show in this section.

An instance of log-HOT PROGRAMMING (log-HP) is an instance $(\mathcal{W}, k)$ of h-hot programming on b address wires when $h = b/2 = \lceil \log_2 n \rceil$. We show that n different nanowires can be addressed with these values of h and b.

Lemma 6.1. *When $h = b/2 = \lceil \log_2 n \rceil$, n different nanowires can be addressed.*

Proof. As shown in the proof of Theorem 3.1, $N(b) = \binom{b}{b/2} = \alpha 2^b/\sqrt{\pi b/2}$, an increasing function of b where $0.97 \leq \alpha \leq 1$. Replacing $b = 2\lceil \log_2 n \rceil$ by $2 \log n$ decreases this function and yields $N(2\lceil \log_2 n \rceil) \geq \alpha n^2/\sqrt{\pi \log_2 n}$, which is an increasing function of n. It has a value greater than n because $\alpha n/\sqrt{\pi \log_2 n} \geq 1$ for $n \geq 2$, which is the desired conclusion. □

Now that the number of address wires needed has been determined, we demonstrate a log-approximation algorithm for log-HP programming similar to the greedy algorithm for d-AP (see the proof of Theorem 6.2).

Theorem 6.4. log-HP *admits a* log*-approximation algorithm under stores alone.*

Proof. We follow the proof of Theorem 6.2 in which an asymptotically continuous reduction is given from d-AP to SET COVER. To establish the reduction from an instance of log-HP to an instance $(U, \mathcal{S}, k)$ of SET COVER we draw a one-to-one correspondence between an h-hot activation matrix of $\mathcal{W}$ and a set cover of U, as shown there. To show that log-HP admits a log-approximation algorithm we need only show that $\mathcal{S}$ can be constructed in time polynomial in the size $(\mathcal{W}, k)$.

As shown in Theorem 6.2, the set U consists of index pairs (i, j) for which $w_{i,j} = 1$. The set $\mathcal{S}$ consists of subsets of U associated with h-hot activation matrices of $\mathcal{W}$.

Since there are b address wires, the number of combinations of columns that can be addressed by activating sets of b or fewer address wires is at most 2^b. Because

$b = 2\lceil \log_2 n \rceil$, it follows that at most $2^{2\lceil \log_2 n \rceil} \leq 2^{\log_2(4n^2)} \leq (4n)^2$ combinations of columns of $\mathcal{W}$ are possible. It follows that the reduction Theorem 6.2 can be done in $O(n(4n)^2)$ comparisons and a number of steps that is polynomial in the size of an instance $(\mathcal{W}, k)$ of log-HP.

It follows that the well-known greedy algorithm for SET COVER provides a log-approximation for log-HP. □

7. The Complexity of *h*-Hot Row Programming

In this section we examine programming of individual rows of an array when h-hot addressing is used. We show that an instance of 1-hot array programming is embedded in 2-hot row programming from which we conclude that 2-hot row programming is **NP**-complete under stores and restores and **NP**-hard to approximate under stores alone unless **P = NP**. We also reduce 2-hot row programming to h-HOT ROW PROGRAMMING, thereby showing that these results carry over to h-HOT ROW PROGRAMMING.

h-HOT ROW PROGRAMMING is the decision problem associated with the optimization problem that seeks to find the smallest number of h-hot operations to program a given row of a nanoarray. Let S be the set of column nanowires that must be addressed to insert 1s into a given row of a nanoarray.

h-HOT ROW PROGRAMMING (h-HRP)
Instance: (S, h, b, k) where the integers in $S \subseteq N = \{1, 2, \ldots, n\}$ are addressed by an h-hot addressing scheme on b address wires, $h < b$ and $n \leq \binom{b}{h}$.
Answer: "Yes" if S can be programmed using at most k h-hot operations.

CLIQUE COVER defined below can be reduced to 2-HOT ROW PROGRAMMING by equating each edge (v, w) with a pair of address wires in 2-hot addressing.:

CLIQUE COVER
Instance: Pairs (G, k) where G is a graph and k is an integer.
Answer: "Yes" if there exists a set of at most k cliques in G such that every edge of G is covered by some clique.

It follows that 2-HOT ROW PROGRAMMING is **NP**-complete under stores alone, since this is true of CLIQUE COVER. Also, when only stores are used, the problem on $n \times n$ arrays is not approximable to within a factor of n^{ε} for some $\varepsilon > 0$ in polynomial time unless **P** $\neq$ **NP**. We obtain stronger results by a reduction from 1-hot array programming to 2-hot row programming, a method that may have promise to simplify row programming.

Theorem 7.1. *There exists an* (a, c)*-bounded reduction from* 1-HOT ARRAY PROGRAMMING *to* 2-HOT ROW PROGRAMMING.

Proof. In an instance of 2-HOT ROW PROGRAMMING each nanowire is addressed by activating two address wires (a_1, a_2). The 1s and 0s in the given row can be placed in a symmetric $b \times b$ array whose axes are labeled by a_1 and a_2 and for which the diagonal

entries are don't cares. (No nanowire is activated by activating one address wire.) The $\lfloor b/2 \rfloor \times \lfloor b/2 \rfloor$ array in the upper right-hand quadrant of this $b \times b$ array is arbitrary. Thus, an instance of 1-HOT ARRAY PROGRAMMING can be solved by embedding it in the upper right-hand quadrant of an instance of 2-HOT ROW PROGRAMMING and its reflection about the diagonal embedded in the lower left-hand quadrant, replacing the other entries with "don't cares" and solving the latter problem. □

The general case of h-HOT ROW PROGRAMMING is **NP**-hard.

Theorem 7.2. *There exists an* (a, c)*-bounded reduction from* 2-HOT ROW PROGRAMMING *to* h-HOT ROW PROGRAMMING.

Proof. Each integer i in the set S of an instance $(S, 2, b, k)$ of 2-HRP is associated with a pair $(a_1^{(i)}, a_2^{(i)})$ of distinct integers, each drawn from a set of b integers. Each integer i in the set S^* of an instance (S^*, h, b^*, k) of h-HRP is associated with an h-tuple $(a_1^{(i)}, a_2^{(i)}, \ldots, a_h^{(i)})$ of distinct integers, each drawn from a set of b^* integers.

We reduce an instance $(S, 2, b, k)$ to an instance of (S^*, h, b^*, k), $b^* = b + h - 2$, by encoding each pair $(a_1^{(i)}, a_2^{(i)})$ associated with an integer in S as an h-tuple $(a_1^{(i)}, a_2^{(i)}, b + 1, b + 2, \ldots, b + h - 1)$ associated with an integer in S^*.

It follows that there exists an activation of at most k address wires that covers all integers in S if and only if there exists an activation of at most k address wires that cover all integers in S^*. □

Since it is obvious that h-HOT ROW PROGRAMMING is in **NP** and the reduction from 2-HOT ROW PROGRAMMING to h-HOT ROW PROGRAMMING is (a, c)-bounded, the following is immediate.

Corollary 7.1. h-HOT ARRAY PROGRAMMING *is* **NP***-complete under stores alone. Furthermore, under stores alone the minimal number of* h*-hot activations to program a row is not approximable to within a factor of* n^{ε} *for some* $\varepsilon > 0$, n *being the number of nanowires, in polynomial time if* $\mathbf{P} \neq \mathbf{NP}$.

8. Heuristic Array Programming

Given the complexity of array programming, it is important to explore heuristics. Experience with other hard problems indicates that good solutions can often be found in practice even when the problem type is known to be **NP**-hard.

In this section we explore heuristics for array programming under h-hot addressing. It may be possible to leverage the results given above for programming structured arrays under 1-hot addressing to program arrays under h-hot addressing.

8.1. *An h-Hot Array Programming Heuristic*

We propose the following general heuristic to program an $n \times n$ array $\mathcal{W}$. Here Φ is the $n \times n$ array of "don't cares." When (de)activation matrices are chosen, they must be

consistent with the h-hot addressing scheme.

let $\mathcal{V} = \mathcal{W}$ and $j = 0$.
while $\mathcal{V} \neq \Phi$,
  let $\mathcal{S}$ be the (de)activation matrix that covers the most 1s (0s)
    in $\mathcal{V}$ when don't cares may be included.
  let $j = j + 1$.
  let $E_j = \mathcal{S}$.
  in $\mathcal{V}$ replace the entries covered by $\mathcal{S}$ by don't cares.

If the sequence of steps $E_1, E_2, \ldots$ is applied in reverse order, the array $\mathcal{W}$ will be generated. Since at least one entry of $\mathcal{V}$ is replaced on each step, the loop in this heuristic will execute at most n^2 times on an $n \times n$ array.

It follows that an efficient heuristic to find the largest subarray of 1s or 0s that may contain don't cares can lead to an efficient heuristic to program an array. However, this problem is equivalent to MAX EDGE WEIGHT BICLIQUE where edges have a weight of 0 or 1. (The 0's correspond to don't cares.) Dawande et al. [6] have shown that the latter problem is NP-complete by asymptotically continuous reduction from MAX CLIQUE. Since the latter problem is hard to approximate within a factor of n^{ε} for some $\varepsilon > 0$ unless **P = NP**, the same is true for the maximum size (de)activation matrix. Thus, it is better to search for an efficient probabilistic heuristic for this problem.

There remains to show how a heuristic for MAX CLIQUE can be used to derive a maximum h-hot (de)activation matrix.

Given an $n \times n$ array $\mathcal{W}$ that is addressed under h-hot, create the graph $\mathcal{G}$ that has a vertex $v_{i,j}$ for each (i, j) for which $w_{i,j} = 1$. Insert an edge between vertices $v_{i,j}$ and $v_{k,l}$ if and only if $w_{i,j}$ and $w_{k,l}$ can be in the same h-hot (de)activation matrix. This is possible when both $w_{i,l}$ and $w_{k,j}$ are either 1 or don't cares. Then a clique on $\mathcal{G}$ corresponds to an activation matrix of $\mathcal{W}$. The reduction when the largest deactivation matrix is sought is almost identical; subarrays of 0s are identified instead of 1s.

8.1.1. *Exploiting strongly structured arrays.* We have demonstrated in Section 4 that strongly structured arrays are relatively inexpensive to program. It follows that identifying structure in arrays allows algorithms similar to those developed in that section to be utilized. Therefore, before applying our heuristic it is desirable to search for similarities to known arrays.

In the absence of known structure, our heuristic for array programming will nevertheless yield highly efficient results when a strong heuristic for MAX EDGE WEIGHT BICLIQUE exists. For example, the heuristic is capable of discovering the optimal program for a banded array even when rows and columns are permuted.

9. Conclusions

In this paper we explore the computational complexity of finding optimal or near optimal solutions to the array programming problem under h-hot addressing. Our results for the general case build on results for the 1-hot model. We show that array programming is

NP-complete when stores and restores are allowed but that it is not possible to find good approximations in polynomial time unless **P = NP** when stores alone are used.

To understand the power of restores, we show that programs for structured prototypical arrays with stores and restores use dramatically fewer steps than programs using stores alone, an improvement that did not seem possible a priori.

Given the difficulty of finding optimal programs, we explore special cases. We show that log-approximate bounds can be found in polynomial time for the following problems: (a) 1-hot programming when the number of 1s (0s) in (de)activation matrices is bounded, (b) 1-hot programming when the number of 1s in each row and column is bounded, and (c) h-hot addressing when h and b are both proportional to $\log n$.

We also examine h-hot row programming and show that it is **NP**-hard under both stores and restores and hard to approximate in polynomial time when only stores are allowed. However, because k-hot array programming reduces to $2k$-hot row programming, fast algorithms for the former may help solve the latter.

Our last topic is a discussion of heuristics to program arrays and rows using heuristics to locate the largest subarray of 1s or 0s in an array of 1s, 0s, and don't cares.

A number of open problems have been identified that are enumerated below:

- Give a good polynomial-time approximation algorithm to the h-hot array programming problem under stores and restores or prove that such an algorithm does not exist unless **P = NP**.
- It has been suggested by André DeHon [7] that the time to write 1s in nanoarrays may be much larger than the time to write 0s. How would this affect the programming of nanoarrays?
- Determine if an efficient algorithm for s-sparse matrices under h-hot addressing can be found.
- Analyze the performance of heuristic algorithms of the type described in Section 8.

Appendix

We prove that instances (G, k), $G = (X \cup Y)$, of CCB in which k is logarithmic in $|X| + |Y|$ is **NP**-hard. It follows that AP is **NP**-hard even if the array can be covered by a number of activation matrices logarithmic in the size of the problem.

Our starting point is to define the notion of a gregarious clique partition, establish its relationship to biclique covers, and apply this relationship to a specific class of sparse graphs. A **biclique cover** is a set of bicliques that includes every edge of G.

GREGARIOUS CLIQUE PARTITION (GCP)
Instance: Pairs (G, k) where $G = (V, E)$ is a graph and k is an integer.
Answer: "Yes" if there exists a set of at most k vertex-disjoint cliques in G such that every vertex of G is included in some clique and for each vertex in a clique there is at most one vertex in each other clique with which it is non-adjacent.

Theorem A.1. *If $G = (V, E)$ has a gregarious clique partition of size c, it also has a biclique cover of size at most $\binom{c}{2}\lceil \log_2 |V| \rceil$.*

Proof. We consider first the case in which G is itself a clique. Without loss of generality, assume that $|V| = 2^p$. (If $|V|$ is not a power of two, embed it in a larger clique that does satisfy this condition and remove the extra vertices and edges later.) We now show that G has a biclique cover of size at most p.

In the base case construct a biclique by partitioning V evenly between sets X and Y and including all edges between these two sets of vertices. This biclique does not cover edges of G that exist between vertices in X nor between vertices in Y, two sets of 2^{p-1} vertices. Split each of X and Y into two sets of equal size, namely, $\{X_1, X_2\}$ and $\{Y_1, Y_2\}$, and form a biclique on the sets $X_1 \cup Y_1$ and $X_2 \cup Y_2$. All edges of G are now covered except for those between the vertices within the 2^2 sets X_1, X_2, Y_1, and Y_2, each of which has 2^{p-2} vertices. Repeat this process by dividing each of these sets into two equal-sized subsets and placing one subset on one side of a bipartite graph and the other subset on the other side and including all edges between them. This results in all edges being covered except for those between vertices in 2^3 sets each of size 2^{p-3}. Repeat this process until all edges in the clique G are covered by some biclique. Clearly, a total of p steps suffice to do this.

Consider the case in which $G = (V, E)$ has a gregarious clique partition containing two cliques $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. We show that it is possible to find a biclique cover of size $\lceil \log_2 |V_3| \rceil$ where $V_3 = V_1 + V_2$.

- If $v_i \in V_1$ has one vertex in V_2 with which it is not adjacent, let $v_{j(i)}$ be that vertex. Form G' from G by coalescing each v_i and $v_{j(i)}$ into v^i as well as coalescing edges (v, v_i) and $(v, v_{j(i)})$ into (v, v^i) for every $v \in (V - \{v_i, v_{j(i)}\})$. Since G contains two cliques, G' is a single clique.
- In each biclique covering G', separate v^i into v_i and $v_{j(i)}$ and split each edge adjacent to v^i thereby converting a biclique cover for G' into one for G.

Given a gregarious clique partition for G of size c, a biclique cover for G is obtained by combining the biclique covers for each pair of cliques in the partition. Since there are $\binom{c}{2}$ pairs, there is a biclique cover of size $\binom{c}{2}\lceil \log_2 n \rceil$ that covers all edges of G. □

THREE EDGE COLORING (3-EC)
Instance: Graphs $G = (V, E)$ of degree three.
Answer: "Yes" if there exists a three-coloring of the edges of G such that no two adjacent edges have the same color.

The following result is due to Holyer [13]. Vizing [26] has shown that a four-coloring of the edges of a graph of degree three can be found in polynomial time.

Lemma A.1. THREE EDGE COLORING *is* **NP**-*complete*.

Let $\mathcal{H}_1$ denote the set of graphs G of maximum degree three. Let $\mathcal{H}_2$ denote the complements of line graphs in $\mathcal{H}_1$, that is, $G' = (V', E')$ is in $\mathcal{H}_2$ if each $v_i \in V'$ corresponds to an edge in E and v_i and $v_j \in V'$ are adjacent if and only if the edges corresponding to v_i and v_j are not adjacent in G.

THREE-CLIQUE PARTITION
Instance: Graphs $G = (V, E)$.
Answer: "Yes" if there exists a three-clique partition of G.

Lemma A.2. THREE-CLIQUE PARTITION *is* **NP**-*complete*.

Proof. Translate $G \in \mathcal{H}_1$ into $G' \in \mathcal{H}_2$. Then G' has a three-clique partition of its vertices if and only if G has a three-coloring of its edges. Furthermore, THREE-CLIQUE PARTITION is in **NP**. □

We now demonstrate that each graph in $\mathcal{H}_2$ has a small gregarious clique partition.

Theorem A.2. *Every graph in* $\mathcal{H}_2$ *has a gregarious clique partition of size at most* 13, *and this partition can be discovered in polynomial time*.

Proof. Let $\mathcal{H}_3$ be the set of graphs G'' constructed from $G \in \mathcal{H}_1$ in which G'' has a vertex v_i for each edge of G and vertices v_i and v_j are connected if and only if the edges of G to which they correspond are not adjacent to each other nor to a common edge.

Let G' in $\mathcal{H}_2$ be obtained from G in $\mathcal{H}_1$. Clearly, a clique partition for G'' is also a clique partition for G'. However, it is actually a gregarious clique partition for G'. To show this, consider vertices v_i and v_j in a common clique of G''. They are in the same clique of G'. If there is no edge in G' between both of them and a vertex v_k in another clique of G', then there is a common edge in G that is adjacent to the two edges corresponding to v_i and v_j. However, this would imply that there cannot be an edge between v_i and v_j in G'', which is a contradiction.

It remains to demonstrate that a clique partition for a graph in $\mathcal{H}_3$ can be found in polynomial time. Since the maximum degree of a graph in $\mathcal{H}_1$ is three, it follows that a vertex of a graph in $\mathcal{H}_3$ is non-adjacent to at most 12 other vertices in that graph. Therefore, the complement of a graph in $\mathcal{H}_3$ has degree 12. For a graph with maximum degree k it is possible in polynomial time to identify a graph coloring of size $k + 1$. Because GRAPH COLORING and CLIQUE PARTITION are identical problems on complementary graphs, it is possible to identify a clique partition of size 13 in polynomial time for any graph in $\mathcal{H}_3$. □

Combining Theorems A.1 and A.2 we have the following lemma.

Lemma A.3. *Every graph* $G = (V, E)$ *in* $\mathcal{H}_2$ *has a biclique cover of size at most* $78\lceil \log_2 |V| \rceil$, *which can be discovered in polynomial time*.

COVERING BY $O(\log n)$ COMPLETE BIPARTITE SUBGRAPHS (CCB-log)
Instance: Pairs (G, k) where $G = (X \cup Y, E)$, $E \subseteq X \times Y$ is a bipartite graph and $k = O(\log(|X| + |Y|))$.
Answer: "Yes" if there exists a set of at most k bicliques of G that covers all the edges of G.

Theorem A.3. CCB-log *is* **NP**-*hard*.

Proof. We give a reduction from THREE-CLIQUE PARTITION to CCB-log.

Given a graph $G = (V, E) \in \mathcal{H}_2$, create a bipartite graph $P = (A \cup B, A \times B)$ with $|A| = |B| = n$ as follows. For each vertex $v_i \in G$, create vertices $a_i \in A$ and $b_i \in B$. Connect a_i to b_i with edge $e_{i,i}$. Connect a_i to b_j with edge $e_{i,j}$ if there is an edge between v_i and v_j.

It follows from this construction that if all the edges in $E_s = \{e_{i,i} \mid 1 \leq i \leq |V|\}$ can be covered with three bicliques that may include edges from $E_d = \{e_{i,j} \mid i \neq j\}$, then a clique partition of size three can be found for G, which is an **NP**-hard problem. However, a partition of size four can always be found [26].

We show that edges E_d in P can be covered with $156\lceil \log_2 n \rceil$ bicliques that do not cover any edges in E_s. We add vertices and edges to P to form the new bipartite graph P'. We show that P' requires exactly $156\lceil \log_2 n \rceil$ bicliques to cover all edges except for those in E_s and that none of the edges in these bicliques cover edges in E_s. Thus, we conclude that P' has a complete bipartite cover of size $156\lceil \log_2 n \rceil + 3$ if and only if G has a clique partition of size three. Furthermore, P' always has a complete bipartite cover of size $156\lceil \log_2 n \rceil + 4$. Finally, P' has $n + 156$ vertices in each of its two sets of vertices.

To show that edges E_d in P can be covered with $156\lceil \log_2 n \rceil$ bicliques that do not cover any edges in E_s we start with a biclique cover for G of size $78\lceil \log_2 n \rceil$ that covers all the edges of G. (See Lemma A.3.) For each biclique $(X \cup Y, X \times Y)$ in this cover, we define two bicliques for the graph P as follows: If $v_i \in X$ and $v_j \in Y$, the first biclique associates $a_i \in A$ with v_i and $b_j \in B$ with v_j. The second associates a_i with v_j and b_j with v_i. Since the biclique cover of G covered all edges in G, these new bicliques will cover all edges in E_d. It is obvious by construction that these bicliques do not cover edges in E_s.

The bipartite graph P' is constructed by adding one vertex a_k to A and one vertex b_k to B for each of the $c \leq 156\lceil \log_2 n \rceil$ bicliques. We also add an edge between a_k and b_k and an edge from a_k (b_k) to each b_j (a_i) in the kth biclique. It follows that exactly c bicliques are needed to cover the edges in E_d as well as the new edges and that these new bicliques do not cover any of the edges in E_s. If $c < 156\lceil \log_2 n \rceil$, add bicliques consisting of one new pair of vertices and an edge between them to bring the number of bicliques up to $156\lceil \log_2 n \rceil$. Clearly, P' can be constructed in polynomial time.

It follows that all edges of P' may be covered by $156\lceil \log_2 n \rceil + 3$ bicliques if and only if edges in E_s can be covered by three bicliques (they will also include edges in E_d). We have already shown that it is **NP**-hard to determine whether edges in E_s can be covered by three bicliques, so it is **NP**-hard to determine whether P may be covered by $156\lceil \log_2 n \rceil + 3$ bicliques. □

Because there is an asymptotically continuous reduction from CCB to AP, the following holds.

Theorem A.4. AP-log *is* **NP**-*complete under stores alone*.

References

[1] W. Arden et al. International technology roadmap for semiconductors, Semiconductor Industry Association, 2001. See http://public.itrs.net.

[2] A. Bachtold, P. Hadley, T. Nakanishi, and C. Dekker. Logic circuits with carbon nanotube transistors. *Nature*, 294:1317–1320, 2001.

[3] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLS Synthesis*. Kluwer, Dordrecht, 1984.

[4] C. P. Collier, E. W. Wong, M. Belohradský, F. M. Raymo, J. F. Stoddart, P. J. Kuekes, R. S. Williams, and J. R. Heath. Electronically configurable molecular-based logic gates. *Science*, 285:391–394, 1999.

[5] Y. Cui, L. Lauhon, M. Gudiksen, J. Wang, and C. M. Lieber. Diameter-controlled synthesis of single crystal silicon nanowires. *Applied Physics Letters*, 78(15):2214–2216, 2001.

[6] M. Dawande, P. Keskinocak, J. M. Swaminathan, and S. Tayur. Multipartite clique problems. *Journal of Algorithms*, 41(2):388–403, November 2001.

[7] A. DeHon. Personal communication, 2003.

[8] A. DeHon, P. Lincoln, and J. E. Savage. Stochastic assembly of sublithographic nanoscale interfaces. *IEEE Transactions on Nanotechnology*, 2(3):165–174, 2003.

[9] C. Dekker. Carbon nanotubes as molecular quantum wires. *Physics Today*, pages 22–28, May 1999.

[10] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1, 3rd edn. Wiley, New York, 1968.

[11] S. Hauck, Z. Li, and E. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1107–1113, 1999.

[12] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS, Boston, MA, 1995.

[13] I. J. Holyer. The NP-completeness of edge coloring. *SIAM Journal on Computing*, 10:718–720, 1981.

[14] Y. Huang, X. Duan, Q. Wei, and C. M. Lieber. Directed assembly of one-dimensional nanostructures into functional networks. *Science*, 291:630–633, 2001.

[15] D. Johnson. Approximate algorithms for combinatorial problems. *Journal of Computer and Systems Sciences*, 9:256–278, 1974.

[16] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.

[17] P. J. Kuekes, R. S. Williams, and J. R. Heath. Molecular wire crossbar memory. Technical Report, US Patent Number 6,128,214, US Patent Office, October 3, 2000.

[18] Z. Li and S. Hauck. Don't care discovery for FPGA configuration compression. *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 91–98, 1999.

[19] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM*, 41(5):960–981, 1994.

[20] N. A. Melosh, A. Boukai, F. Diana, B. Gerardot, A. Badolato, P. M. Petroff, and J. R. Heath. Ultrahigh-density nanowire lattices and circuits. *Science*, 300:112–115, 2003.

[21] A. M. Morales and C. M. Lieber. A laser ablation method for synthesis of crystalline semiconductor nanowires. *Science*, 279:208–211, 1998.

[22] M. A. Reed and J. M. Tour. Computing with molecules. *Scientific American*, 282(86):86–93, 2000.

[23] T. Rueckes, K. Kim, E. Joselevich, G. Y. Tseng, C.-L. Cheung, and C. M. Lieber. Carbon nanotube-based nonvolatile random access memory for molecular computing. *Science*, 289:94–97, 2000.

[24] H. U. Simon. On approximate solutions for combinatorial optimization problems. *SIAM Journal on Discrete Mathematics*, 3(2):294–310, May 1990.

[25] L. J. Stockmeyer. The set basis problem is NP-complete. Technical Report RC-5431, IBM, 1975.

[26] V. G. Vizing. On an estimate of the chromatic class of a p-graph. *Diskretnyi Analiz*, 3:23–20, 1964.

Received August 24, 2003, *and in final form March* 30, 2004. *Online publication February* 9, 2005.