PAPER *Special Section on Recent Advances in Circuits and Systems*

Hardware n Choose k Counters with Applications to the Partial Exhaustive Search

Koji NAKANO[†a)], ***Member and*** **Youhei YAMAGISHI**[†], ***Student Member***

SUMMARY The main contribution of this work is to present several hardware implementations of an "n choose k" counter ($\mathrm{C}(n,k)$ counter for short), which lists all n-bit numbers with $(n-k)$ 0's and k 1's, and to show their applications. We first present concepts of $\mathrm{C}(n,k)$ counters and their efficient implementations on an FPGA. We then go on to evaluate their performance in terms of the number of used slices and the clock frequency for the Xilinx VirtexII family FPGA XC2V3000-4. As one of the real life applications, we use a $\mathrm{C}(n,k)$ counter to accelerate a digital halftoning method that generates a binary image reproducing an original gray-scale image. This method repeatedly replaces an image pattern in small square regions of a binary image by the best one. By the partial exhaustive search using a $\mathrm{C}(n,k)$ counter we succeeded in accelerating the task of finding the best image pattern and achieved a speedup factor of more than 2.5 over the simple exhaustive search.

key words: *FPGA-based computing, instance-specific solutions, digital halftoning*

1. Introduction

An FPGA (Field Programmable Gate Array) is a programmable VLSI in which a hardware design can be embedded quickly. Typical FPGAs consist of an array of programmable logic elements, distributed memory blocks, and programmable interconnections between them. The logic block usually contains either a four-input logic function or a multiplexer and several flip-flops. The distributed memory block is usually a dual-port RAM on which a word of data for possibly distinct addresses can be read/written at the same time. Design tools are available to the users to embed their hardware logic designs into the FPGAs. Our goal is to use FPGAs to accelerate useful computations. In particular, it is very challenging to develop FPGA-based solutions that are faster and more efficient than traditional software solutions.

Let $\mathrm{C}(n,k)$ denote a set of all n-bit binary numbers that has $(n-k)$ 0's and k 1's. For example, $\mathrm{C}(6,3)$ is

$$\begin{aligned}\mathrm{C}(6,3) = \{&000111, 001011, 001101, 001110, 010011,\\ &010101, 010110, 011001, 011010, 011100, 100011,\\ &100101, 100110, 101001, 101010, 101100, 110001,\\ &110010, 110100, 111000\}. \end{aligned} \tag{1}$$

An *"n choose k" counter* ($\mathrm{C}(n,k)$ counter for short) is a counter that lists all numbers in $\mathrm{C}(n,k)$. The main contribution of this paper is to present several hardware implementations of $\mathrm{C}(n,k)$ counter. We first present the concept of $\mathrm{C}(n,k)$ counters and discuss several straightforward implementations on an FPGA. We then go on to present several efficient implementations of $\mathrm{C}(n,k)$ counters on an FPGA.

The second contribution of this paper is to use a $\mathrm{C}(n,k)$ counter to accelerate a digital halftoning method [6], which repeats the partial exhaustive search. digital halftoning is a key operation to obtain binary images for printing [7], [8]. We use the partial exhaustive search to reduce the search space of the exhaustive search performed by a digital halftoning method presented in [6]. We have developed a halftoning system using a PCI-connected FPGA board with a Xilinx VirtexII family FPGA, XC2V3000-4 [15]. By the partial exhaustive search using a $\mathrm{C}(n,k)$ counter, we have achieved a speedup factor of 2.5 to 4.0 for digital halftoning.

This paper is organized as follows. In Sect. 2, we show a concept of $\mathrm{C}(n,k)$ counters and motivation of our research. We then discuss several straightforward implementations of $\mathrm{C}(n,k)$ counters in Sect. 3. In Sect. 4, we show basic ideas for efficient implementation of $\mathrm{C}(n,k)$ counters. Sections 5 and 6 present the details of our implementations of a $\mathrm{C}(n,k)$ counter. In Sect. 7, we evaluate the performance of these implementations using Xilinx VirtexII FPGA, XC2V3000-4. Section 8 shows how we apply the $\mathrm{C}(n,k)$ counters to the digital halftoning. Section 9 offers concluding remarks.

2. Concept and Motivation for C(n, k) Counters

It is well known that an n-bit binary counter can be simply implemented using n DFFs (D-type Flip Flops) and n HAs (Half Adders). A binary counter is mainly used to enumerate the number of events, which are represented as edge triggers of a signal. On the other hand, an n-bit binary counter can also be used to list all 2^n binary numbers. For example, suppose that we have a function $f : \{0,1\}^n \rightarrow \{0,1,\ldots,m\}$ for some positive integer m, and we need to find an n-bit binary number $\boldsymbol{r}$ such that $f(\boldsymbol{r})$ takes the minimum value over all possible 2^n n-bit binary numbers $\boldsymbol{x}$. In other words, our task is to compute

$$\boldsymbol{r} = \arg\min_{\boldsymbol{x}\in\{0,1\}^n} f(\boldsymbol{x}). \tag{2}$$

This task is a kind of combinatorial optimization, which has many practical applications. A fast and efficient solution for

Manuscript received September 29, 2004.
Manuscript revised February 5, 2005.

[†]The authors are with the School of Engineering, Hiroshima University, Higashi-hiroshima-shi, 739–8527 Japan.
a) E-mail: nakano@fse.hiroshima-u.ac.jp

DOI: 10.1093/ietisy/e88–d.7.1350

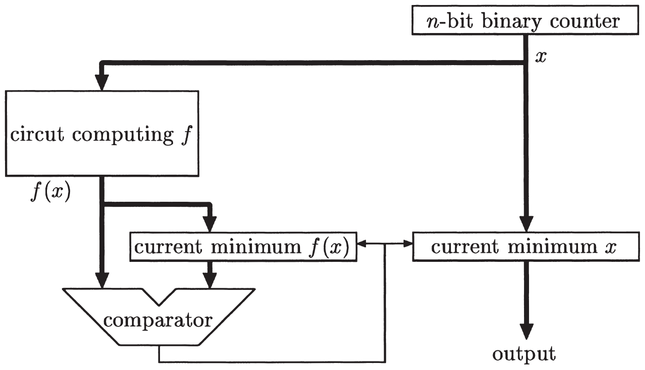


Fig. 1 Illustrating the hardware for computing $\arg\min_{\boldsymbol{x}\in\{0,1\}^n} f(\boldsymbol{x})$.

this task is to design an *instance-specific solution* using an FPGA as follows. We design a circuit that computes $f(\boldsymbol{x})$ for any given n-bit binary numbers $\boldsymbol{x}$. The output $\boldsymbol{x}$ of the n-bit counter is given to this circuit computing $f(\boldsymbol{x})$. A comparator is used to compare the current value of $f(\boldsymbol{x})$ and the minimum value obtained so far. If the current value $f(\boldsymbol{x})$ is smaller, then the current minimum $f(\boldsymbol{x})$ and $\boldsymbol{x}$ are updated. We refer the reader to Fig. 1 for an illustration of the hardware computing formula (2). This hardware approach is promising whenever there exists an efficient (i.e. compact and of small depth) circuit computing f. An example of function f for which this approach works efficiently is the MAX-SAT problem. An input instance of the MAX-SAT problem is a set of m Boolean formulas $f_1, f_2, \ldots, f_m$ of n Boolean variables. MAX-SAT problem is a combinatorial optimization problem to find an assignment of Boolean variable values that maximizes the number of satisfied formulas (or minimizes the number of unsatisfied formulas). To solve the MAX-SAT problem using the above approach, we define function $f : \{0, 1\}^n \to \{0, 1, \ldots, m\}$ such that

$$f(\boldsymbol{x}) = |\{f_i | f_i(\boldsymbol{x}) \text{ is not satisfied}\}|. \tag{3}$$

It should be clear that, $\boldsymbol{r}$ in formula (2) for function f in (3) is an optimal solution of the MAX-SAT problem. Also, Boolean formulas can be implemented in the FPGA by a combinational circuit in an obvious way. For example, an AND binary operator in a Boolean formula can be implemented using AND gates with fan-in 2. Thus, the circuit computing $f(\boldsymbol{x})$ above can be implemented in the FPGA very efficiently and the above approach works for the MAX-SAT problem. This approach is an instance-specific solution [1], [13], [19], because the circuit embedded in the FPGA depends on the input instance (i.e. m Boolean formulas) of the problem.

The above approach is also called (simple) *exhaustive search*, which has a quite large search space of all 2^n values. This approach is not practical even if n is not large, say, n = 40. So, many researchers have devoted to develop practical methods to solve this type of combinatorial optimization problem. For example, heuristic approaches such as local search, genetic algorithms, approximation algorithm, and randomized algorithm are used to find either nearly optimal solution or the best solution with high probability [4], [11]. Also, several FPGA-based instance-specific approaches for solving SAT problem have been presented [17]–[19]. This paper presents a different approach that we call *partial exhaustive search*, This approach reduces the size of the search space.

Sometimes, function f has some properties which enable us to reduce the size of the search space. Let us see some examples. The first example is a property of *biased input*. Suppose that an input instance of the MAX-SAT is given as a CNF(Conjunctive Normal Form) and most of the literals in the input formula are negative. If this is the case, it is expected that the optimal solution has few 1 (or true) assignments. Hence, we can omit the evaluation of the value of $f(\boldsymbol{x})$ for input $\boldsymbol{x}$ that has many 1's.

Another example of the properties for function f that enables us to reduce the size of the search space is a property of *concavity*. Let $\boldsymbol{r}_k$ be the optimal solution of $f(\boldsymbol{x})$ over all numbers in $\mathrm{C}(n, k)$, that is,

$$\boldsymbol{r}_k = \arg\min_{\boldsymbol{x}\in \mathrm{C}(n,k)} f(\boldsymbol{x}). \tag{4}$$

It should be clear that

$$\boldsymbol{r} = \arg\min\{f(\boldsymbol{r}_k) \mid 0 \le k \le n\}. \tag{5}$$

A function f is *concave* if there exists i ($1 \le i \le n$) such that

$$\begin{aligned} &f(\boldsymbol{r}_0) \ge f(\boldsymbol{r}_1) \ge \cdots \ge f(\boldsymbol{r}_{i-1}) \ge \\ &\quad f(\boldsymbol{r}_i) \le f(\boldsymbol{r}_{i+1}) \le \cdots \le f(\boldsymbol{r}_n). \end{aligned} \tag{6}$$

Clearly, if f is concave and satisfies the above relation, then $f(\boldsymbol{r}) = f(\boldsymbol{r}_i)$ and $\boldsymbol{r}_i$ is an optimal solution. If f is concave, we can find $\boldsymbol{r}_i$ by the binary search or linear search techniques on $f(\boldsymbol{r}_0), f(\boldsymbol{r}_1), \ldots, f(\boldsymbol{r}_n)$. Hence, we do not have to evaluate f over all 2^n n-bit numbers. Since the exhaustive search is performed to compute $f(\boldsymbol{r}_k)$ for each k, we call this approach the partial exhaustive search.

If function f satisfies these properties, it is sufficient to compute $\boldsymbol{r}_k$ in (4) for several k's. To compute $\boldsymbol{r}_k$ by the instance-specific FPGA-based approach we can use the hardware illustrated in Fig. 1, where an n-bit binary counter is replaced by a $\mathrm{C}(n, k)$ counter. Thus, it is significant work to design efficient implementation of $\mathrm{C}(n, k)$ counters.

3. Straightforward Implementations of $C(n, k)$ Counters

As we have mentioned, it is well known that an n-bit binary counter can be implemented using n DFFs and n HAs. However, the implementation of a $C(n, k)$ counter is not trivial.

We classify implementations of a $C(n, k)$ counter using the following terminology:

lexicographical : an implementation of a $C(n, k)$ counter is *lexicographical* if it outputs all numbers in lexicographical order. More precisely, in lexicographical order, for any two $C(n, k)$ numbers $\boldsymbol{x}$ and $\boldsymbol{y}$, $\boldsymbol{x}$ must appear before $\boldsymbol{y}$ if $\boldsymbol{x} < \boldsymbol{y}$.

redundant : an implementation of a $C(n, k)$ counter is *redundant* if it outputs more than $\binom{n}{k}$ numbers including all numbers in $C(n, k)$. A redundant implementation must provide *a redundant bit* indicating that the current output is redundant. In other words, the redundant bit is low (or 0) in exactly $\binom{n}{k}$ clock cycles and every number in $C(n, k)$ is provided in these clock cycles.

If an implementation of a $C(6, 3)$ counter outputs all numbers in (1) in this order it is lexicographical. It is also non-redundant if all the 20 6-bit numbers in (1) are provided one by one in every clock cycle. Sometimes, lexicographical implementation of $C(n, k)$ is necessary, because the lexicographically first best solution is required in some combinatorial optimization problems [10]. All of the implementations presented in this paper are lexicographical.

Let us observe a simple example of a redundant implementation of a $C(n, k)$ counter that we call *the naive implementation.* This implementation uses an n-bit binary counter and a tree of adders. The output sequence of the naive implementation for $C(4, 2)$ counter is as follows:

$$0000[1], 0001[1], 0010[1], 0011[0], 0100[1], 0101[0],$$
$$0110[0], 0111[1], 1000[1], 1001[0], 1010[0], 1011[1],$$
$$1100[0], 1101[1], 1110[1], 1111[1],$$

where [0] and [1] represent the redundant bit. The redundant bit is 1 iff the number of 1's in a 4-bit number is not 2. The n-bit output is exactly the output of an n-bit binary counter. The Muller-Preparata's circuit [9], [12]–[14], which enumerates the number of 1's in an n-bit binary number, is used to compute the number of 1's in the current output. The basic structure of the Muller-Preparata's circuit is a tree of adders, which has $O(n)$ gates with depth $O(\log n)$ [14]. We can determine if the current output has exactly k 1's using an $\log n$-bit comparator, and so the redundant bit can be provided in an obvious way. However, this naive implementation has too many redundant numbers. If we use it for the partial exhaustive search, the search operation for all possible 2^n instances has to be performed.

Another simple implementation of $C(n, k)$ is *the ROM implementation*, which uses a memory block of an FPGA as a ROM. In the ROM implementation, we use an n-bit 2^n-word ROM, which stores all numbers in $C(n, 0)$, $C(n, 1)$, ..., $C(n, n)$. More specifically, the j-th ($0 \leq j \leq \binom{n}{k} - 1$) number of $C(n, k)$ is stored in a word with address $C(n, 0) + C(n, 1) + \cdots + C(n, k-1) + j$. It should be clear that, by reading words with addresses $\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{k-1}$ to $\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{k} - 1$, we can obtain all numbers in $C(n, k)$. Note that $2^n = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n}$. The ROM implementation is possible only if a ROM which can store all necessary 2^n n-bit numbers is available. Since current FPGAs have memory blocks of up to several Mega bits, $C(n, k)$ implementation is possible if n is small, say $n = 16$. If $n = 16$, we need $16 \cdot 2^{16} = 1$M bits.

In this paper, we focus on implementations of a $C(n, k)$ counter which does not use memory blocks of FPGA. We first present two non-redundant implementations of a $C(n, k)$ counter that we call *the simple shift* and *the binary shift.* The simple shift implementation runs in high frequency for small n although it uses so many gates that it does not fit in the FPGA for large n. The binary shift implementation uses much smaller number of gates, but it runs in low frequency. We then go on to present two redundant implementations of a $C(n, k)$ counter that we call *the left shift* and *the right shift.* The key idea of these implementations is to use the shift register to find the next number. These implementations work in higher frequency than non-redundant implementations for large n. The right shift implementation has fewer redundant states than the left shift implementation if $k > \frac{n}{2}$ and has more if $k < \frac{n}{2}$. If $k = \frac{n}{2}$, they have the same number of redundant states.

Table 1 summarizes the theoretical analysis of the used ROM bits, the number of used gates, the maximum delay between DFFs, and the clock cycles necessary to lists all $\binom{n}{k}$ numbers in $C(n, k)$. Note that an implementation is redundant if it runs in more than $\binom{n}{k}$ clock cycles. Every implementation uses n DFFs to store a current n-bit number and $O(\log n)$ DFFs for storing the value of k and for the state control. The naive and the ROM implementations use a binary counter involving an n-bit adder, which can be implemented in $O(n)$ gates with $O(\log n)$ depth using the carry lookahead technique [2], [16]. The ROM implementation uses a ROM with $n2^n$ bits. The details of the theoretical analysis of our implementations will be given later. Although theoretical analysis is important for large n, it often does not reflect real performance for practically small n. Hence, we evaluate the clock frequency and the size of used hardware resource of an FPGA.

4. Basic Ideas for Implementing $C(n, k)$ Counters

The main purpose of this section is to show basic ideas for implementing non-trivial $C(n, k)$ counters.

We can list all numbers in $C(n, k)$ in lexicographical order by the following five rules:

Rule 0: (initialization) Let the current number be $0^{n-k}1^k$.
Rule 1: If the current number is $(0+1)^*010^i$ for some $i \geq 0$,

Table 1 Theoretical analysis of the performance of implementations of C(n, k) counters.

implementations	DFFs	ROM bits	gates	delay	clock cycles
naive	$n + O(\log n)$	0	$O(n)$	$O(\log n)$	2^n
ROM	$n + O(\log n)$	$n2^n$	$O(n)$	$O(\log n)$	$\binom{n}{k}$
simple shift	$n + O(\log n)$	0	$O(n^2)$	$O(\log n)$	$\binom{n}{k}$
binary shift	$n + O(\log n)$	0	$O(n)$	$O(\log n)$	$\binom{n}{k}$
right shift	$2n + O(\log n)$	0	$O(n)$	$O(\log n)$	$\binom{n}{k} + \binom{n-1}{k} - (n-k)$
left shift	$2n + O(\log n)$	0	$O(n)$	$O(\log n)$	$\binom{n}{k} + \binom{n-1}{k-1} - k$

Table 2 Examples of x, y, z, u, s, and t.

i	10	9	8	7	6	5	4	3	2	1
x(current)	0	1	1	0	1	1	1	0	0	0
y	–	1	0	0	1	0	0	0	0	0
z	–	1	1	1	1	0	0	0	0	0
u	–	0	0	0	1	0	0	0	0	0
s	–	–	0	0	0	1	1	0	0	0
t	–	–	0	0	0	0	0	0	1	1
x(next)	0	1	1	1	0	0	0	0	1	1

then the next number is $(0 + 1)^*100^i$.

Rule 2: If the current number is $(0+1)^*011^i$ for some $i \geq 1$, then the next number is $(0 + 1)^*101^i$.

Rule 3: If the current number is $(0 + 1)^*011^j0^i$ for some $i \geq 1$ and $j \geq 1$, then the next number is $(0 + 1)^*100^i1^j$.

Rule 4: (termination) If the current number is 1^k0^{n-k}, then the listing is terminated.

Note that, as used in regular expressions, $(0 + 1)^*$ represents any sequence over $\{0, 1\}$ of length zero or longer, and 1^k represents a sequence of consecutive k 1's.

The key rules are Rules 1, 2, and 3. Let us see how the next number is determined. Let $x_n x_{n-1} \cdots x_1$ be the current number. Further, let p $(1 \leq p \leq n - 1)$ be the smallest index of x such that $x_{p+1} = 0$ and $x_p = 1$. The next number can be obtained using the following two operations:

swap operation swap the values of x_{p+1} and x_p.
shift operation shift $x_{p-1}x_{p-2} \cdots x_1$ to the right until $x_1 = 1$.

In Rules 1 and 2, the swap operation is performed to find the next number. Both the swap and the shift operations are performed when Rule 3 is applied.

First, we show how we implement the swap operation which is performed in Rules 1, 2, and 3. For this purpose, we determine index p above. Let $y_i = \overline{x_{i+1}} \wedge x_i$ for every i $(1 \leq i \leq n - 1)$. Further, let $z_i = y_i \vee y_{i-1} \vee \cdots \vee y_1$, for every i $(2 \leq i \leq n)$. Since z is the prefix OR of y, z can be obtained using the parallel prefix circuit [2], [3], which has $O(n)$ gates of depth $O(\log n)$. Let $u_1 = z_1$, and $u_i = z_i \wedge \overline{z_{i-1}}$ for each $(2 \leq i \leq n - 1)$. It should be clear that, $u_i = 1$ iff $p = i$. We refer the reader to Table 2 for examples of x, y, z, and u. The swap operation can be simply done by

$$x_i \leftarrow x_i \oplus (u_i \vee u_{i-1}) \qquad (1 \leq i \leq n), \tag{7}$$

where $u_n = u_0 = 0$ and $\oplus$ denotes the XOR operator.

Next, we will show how the shift operation is implemented. Recall that the shift operation is performed for Rule 3. Let $s_i = \overline{z_i} \wedge x_i$ for each i $(1 \leq i \leq n - 2)$. Clearly, s is a sequence of bits to be shifted to the right. Let $t_{n-2}t_{n-3} \cdots t_1$ be a sequence of bits that can be obtained by repeating the shift of $s_{n-2}s_{n-3} \cdots s_1$ until the rightmost bit is 1. We refer the reader to Table 2 for examples of s and t. Once t is obtained, we can perform the shift operation by the following formula:

$$x_i \leftarrow (x_i \wedge z_i) \vee t_i \qquad (1 \leq i \leq n), \tag{8}$$

where $s_n = s_{n-1} = t_n = t_{n-1} = 0$ for simplicity. We assume that every bit of t_i is 0 when all bits of s_i are 0. Then, when Rules 1 or 2 are applied, $s_i = t_i = 0$ for all i. Thus, from formulas (7) and (8) combined, regardless of the applied rules, the next number x can be obtained by a single formula as follows:

$$x_i \leftarrow ((x_i \oplus (u_{i-1} \vee u_i)) \wedge z_i) \vee t_i. \tag{9}$$

Note that if $z_{n-1} = 0$ then $y_{n-1} = y_{n-2} = \cdots = y_1 = 0$. If this is the case, there exists no p such that $x_{p+1} = 0$ and $x_p = 1$. In other words, $x_n x_{n-1} \cdots x_1 = 1^k0^{n-k}$ and Rule 4 (termination) should be applied.

As we have seen, y can be obtained by $n - 1$ NOT gates and $n - 1$ AND gates. The prefix OR circuit, which can be implemented using $O(n)$ gates of depth $O(\log n)$ [2], is used to compute z. Once z is obtained, u and s can be computed using $n-2$ NOT gates and $n-2$ AND gates, each. After that, if t is obtained, each x_i can be computed using two OR gates, one AND gate, and one XOR gate. Thus, a C(n, k) counter can be implemented using $O(n)$ gates of depth $O(\log n)$ excluding the circuit for computing t from s. However, it is not easy to obtain t. In what follows, we will show how we obtain t from s. For later reference, let $s = 0^{N-l-m}1^l0^m$, where $N = n - 2$. Clearly, we need to compute $t = 0^{N-l}1^l$.

5. Non-redundant Implementations of C(n, k) Counters

The main purpose of this section is to show two implementations *the simple shift* and *the binary shift* that compute t from s in a clock cycle. Thus, these implementations for a C(n, k) counter are non-redundant.

5.1 The Simple Shift Implementation

The simple shift implementation uses all the shifted sequences of s. For each i and j $(1 \leq i \leq N; 0 \leq j \leq N - 1)$, let

$$s_i^{[j]} = s_{i+j} \qquad \text{if } i + j \leq N$$

$$= 0 \qquad \text{if } i + j > N. \tag{10}$$

In other words, $s^{[j]}$ is a sequence obtained by shifting s by j bits to the right. Then, t can be obtained by

$$t_i = (s_1^{[0]} \wedge s_i^{[0]}) \vee (s_1^{[1]} \wedge s_i^{[1]}) \vee \cdots \vee (s_1^{[N-1]} \wedge s_i^{[N-1]}). \tag{11}$$

Let us confirm that t is correctly computed by formulas (10) and (11). Recall that $s = 0^{N-l-m}1^l0^m$. Thus, $s_i^{[j]} = 1$ iff $m + 1 \leq i + j \leq m + l$. Since $s_1^{[0]} = s_1^{[1]} = \cdots = s_1^{[m-1]} = 0$, $s_1^{[m]} = s_1^{[m+1]} = \cdots = s_1^{[m+l-1]} = 1$, and $s_1^{[m+l]} = s_1^{[m+l+1]} = \cdots = s_1^{[N-1]} = 0$, we have $t_i = s_i^{[m]} \vee s_i^{[m+1]} \vee \cdots \vee s_i^{[m+l-1]}$. Hence, $t_i = 1$ iff $[m + i, m + l - 1 + i] \cap [m + 1, m + l]$ is not empty, that is $1 \leq i \leq l$. Therefore, $t_1 = t_2 = \cdots = t_l = 1$ and $t_{l+1} = t_{l+2} = \cdots = t_N = 0$, and thus t is computed correctly. Let us evaluate the number of gates used to compute t. Since $s_i^{[N-i+1]} = s_i^{[N-i+2]} = \cdots = s_i^{[N-1]} = 0$ $(i \geq 2)$ always holds, t_i can be computed using $N - i + 1$ AND gates and $N - i$ OR gates. Thus, t can be computed using at most $N + (N - 1) + \cdots + 1 < \frac{N(N+1)}{2} < \frac{n^2}{2}$ AND gates and at most $(N - 1) + (N - 2) + \cdots + 1 < \frac{N^2}{2} < \frac{n^2}{2}$ OR gates. Since each t_i can be computed by a tree of $N - 1$ OR gates with fan-in 2, the depth of the circuit is $O(\log N) = O(\log n)$.

5.2 The Binary Shift Implementation

The binary shift implementation computes the binary representation of the number of 1's in s and generates the same number of 1's by exponential shifting. For simplicity, we assume that $N = 2^u - 1$ for some integer u. Let l be the number of 1's in s and $l_u l_{u-1} \cdots l_1$ be the binary representation of l, that is $l = l_u \cdot 2^{u-1} + l_{u-1} \cdot 2^{u-2} + \cdots + l_1 \cdot 2^0$. The binary representation $l_u l_{u-1} \cdots l_1$ can be computed by the Muller-Preparata's adder tree circuit [12]. Let $s^{\langle j \rangle}$ $(0 \leq j \leq u)$ be a sequence of length $2^j - 1$ determined by the following procedure[†].

```
for $j \leftarrow 1$ to $u$ do
    if $l_j = 0$ then $s^{\langle j \rangle} \leftarrow 0^{2^{j-1}} s^{\langle j-1 \rangle}$
    else $s^{\langle j \rangle} \leftarrow s^{\langle j-1 \rangle} 1^{2^{j-1}}$
```

If $l_j = 1$ then 2^{j-1} 1's are added to the sequence. Thus, it is not difficult to see that $t = s^{\langle u \rangle}$ holds. Further, each $s^{\langle j \rangle}$ can be computed from $s^{\langle j-1 \rangle}$ using $2^j - 1$ multiplexers whose output is determined by l_j. Thus, t can be computed using at most $2^1 - 1 + 2^2 - 1 + \cdots + 2^u - 1 < 2N < 2n$ multiplexers. Also, it is easy to confirm that the depth of the circuit is $O(\log n)$.

6. Redundant Implementations of C(n, k) Counters

In this section, we present two implementations called *right shift* and *left shift* that compute t from s in several clock cycles. Since we do not have to compute t in a single clock cycle, we can obtain high clock frequency. The key idea is to (cyclically) shift s by one position either to the right or to the left until we obtain t. Both implementations use an N-bit shift register.

6.1 The Right Shift Implementation

The right shift implementation uses an N-bit shift register to store s and shift it by one to the right in every clock cycle. Again, recall that $s = 0^{N-l-m}1^l0^m$. Thus, it takes m clock cycles to obtain s. Using the right shift implementation of a C(6, 3) counter, we have the following output sequences:

$$0\overline{01}11[0], 001\overline{01}1[0], 0011\overline{01}[0], 0\overline{01}110[0],$$
$$010\hat{1}\hat{1}0[1], 010\overline{01}1[0], 0101\overline{01}[0], 01\overline{01}10[0],$$
$$0110\hat{1}0[1], 0110\overline{01}[0], 011\overline{01}0[0], \overline{01}1100[0],$$
$$10\hat{1}\hat{1}00[1], 100\hat{1}\hat{1}0[1], 100\overline{01}1[0], 1001\overline{01}[0],$$
$$1\overline{00}110[0], 1010\hat{1}0[1], 1010\overline{01}[0], 101\overline{01}0[0],$$
$$1\overline{01}100[0], 110\hat{1}00[1], 1100\hat{1}0[1], 1100\overline{01}[0],$$
$$110\overline{01}0[0], 11\overline{01}00[0], 111000[0],$$

where [1] and [0] denote the value of the redundant bit, $\overline{01}$ are two bits where the swap operation will be performed, and $\hat{1}$ is a bit 1 where the shift operation is performed. Note that, when Rule 3 is applied, the swap operation is performed before the shift operation starts. This example has 7 redundant states, in which the redundant bit is high ([1]).

Let us evaluate the number of redundant states of a C(n, k) counter using the right shift implementation. For this purpose, let us observe the numbers that appear in the redundant state. Such numbers for the C(6, 3) counter are as follows:

$$010110, 011010, 101100, 100110, 101010, 110100,$$
$$110010.$$

The numbers in the redundant state satisfy the following properties:

(1) the rightmost bit is 0,
(2) three (i.e k) 1's are not consecutive, and
(3) every number satisfying (1) and (2) appears exactly once.

If the rightmost bit of a number is 1, then the shift operation is completed and it is not a redundant state. Hence (1) must be satisfied. Also, a number that has consecutive k 1's cannot be a number in the redundant state, because the swap operation is performed before the shift operation starts. Thus, (2) must be satisfied. If a number satisfying (1) and (2) appears twice, then the same number appear after the shift operation is completed. Since this is not possible, no number satisfying (1) and (2) appears twice. Further, every number satisfying (1) and (2) must appear, and thus (3) must be satisfied.

It is not difficult to confirm that, $\binom{n-1}{k}$ numbers satisfy (1), and $n - k$ numbers have consecutive k 1's among them. Thus, we have,

[†]Note that $s^{\langle 0 \rangle}$ is the null string of length zero.

Theorem 1: The right shift implementation of a $C(n,k)$ counter has $\binom{n-1}{k} - (n-k)$ redundant states.

Since we have $\binom{n}{k}$ non-redundant states, the ratio of the redundant and the non-redundant states is approximately

$$\frac{\binom{n-1}{k} - (n-k)}{\binom{n}{k}} \approx \frac{n-k}{n}.$$

Therefore, the right shift implementation is efficient for larger k.

6.2 The Left Shift Implementation

In the left shift implementation, we shift the register storing s cyclically by one position to the left in every clock cycle. For example, if $s = 0011100$, then the cyclic shift is performed as follows:

$$0011001, 0010011, 0000111.$$

The left shift implementation compute $t = 0^{N-l}1^{l}$ from $s = 0^{N-l-m}1^{l}0^{m}$ in l clock cycles. Using the left shift implementation of a $C(6,3)$ counter, we have the following output sequences:

$$\begin{aligned}
&00\overline{01}11[0], 001\overline{01}1[0], 0011\overline{01}[0], 001\hat{1}\hat{1}0[0],\\
&001\hat{1}0\hat{1}[1], 0\overline{01}011[1], 010\overline{01}1[0], 0101\overline{01}[0],\\
&0101\hat{1}0[0], 01\overline{01}01[1], 0110\overline{01}[0], 011\overline{01}0[0],\\
&01\hat{1}\hat{1}00[0], 01\hat{1}00\hat{1}[1], \overline{01}0011[1], 100\overline{01}1[0],\\
&1001\overline{01}[0], 1001\hat{1}0[0], 10\overline{01}01[1], 1010\overline{01}[0],\\
&101\overline{01}0[0], 101\hat{1}00[0], 1\overline{01}001[1], 1100\overline{01}[0],\\
&110\overline{01}0[0], 11\overline{01}00[0], 111000[0].
\end{aligned}$$

This example also has 7 redundant states. To simplify the evaluation of the number of redundant states, we assume that the swap operation is performed after the shift operation is completed.

Let us evaluate the number of the redundant states. Again, let us observe the numbers that appear in the redundant states using those for $C(6,3)$ as follows:

$$001101, 001011, 010101, 011001, 010011, 100101,$$
$$101001.$$

Similarly, these numbers in the redundant state satisfy the following properties:

(1) the rightmost bit is 1,
(2) three (i.e $n-k$) 0's are not consecutive, and
(3) every number satisfying (1) and (2) appears exactly once.

Since we can verify these properties similarly to those for the right shift implementation, we omit the proof. It is easy to see that $\binom{n-1}{k}$ numbers satisfy (1), and k numbers have consecutive $n-k$ 0's among them. Thus we have,

Theorem 2: The left shift implementation of a $C(n,k)$ counter has $\binom{n-1}{k-1} - k$ redundant states.

The ratio of the redundant and the non-redundant states for the left shift implementation is approximately

$$\frac{\binom{n-1}{k-1} - k}{\binom{n}{k}} \approx \frac{k}{n}.$$

Therefore, for small k, the left shift implementation has fewer redundant states than the right shift implementation.

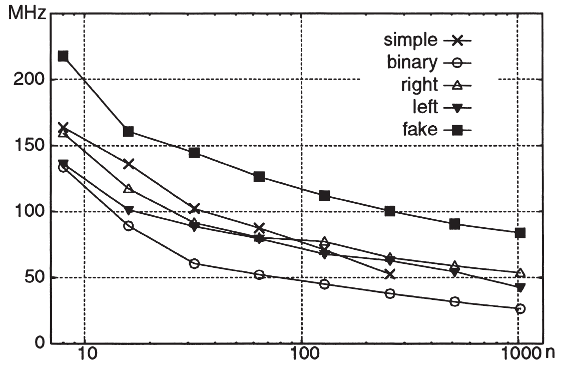
We should note that by inverting the output of a $C(n, n-k)$ counter using n NOT gates, we can obtain a $C(n,k)$ counter. Thus, we can obtain a $C(n,k)$ counter with fewer redundant states using the right shift implementation for small k. However, the resulting output sequence is not lexicographical.

7. Performance Evaluation

This section is devoted to show the performance evaluation for the Xilinx VirtexII family FPGA XC2V3000-4, which has 14336 slices. A slice is a unit block of the VirtexII FPGA, which has two four-input function generators, carry logic, multiplexers, and two storage elements [5]. We have used Xilinx ISE logic design tool (Ver 6.3i) to analyze the timing and the number of slices used. We have wrote the HDL source codes for $C(n,k)$ counter implementations in RTL (Register Transfer Level) of Verilog HDL. We have used default parameter values, for example "Optimization goal = Speed" and "Optimization effort = Normal", for logic synthesis using Xilinx ISE logic design tool. Also, we gave no user constraints to synthesize our Verilog HDL source codes.

Figure 2 shows the clock frequency and the number of used slices for $n = 8, 16, 32, 64, 128, 256, 512$, and 1024 estimated based on the net list obtained by XST logic synthesis tool, which is a part of Xilinx ISE logic design tool. Note that, the performance are obtained from the net list. After the implementation (i.e. mapping and routing), the actual clock frequency can be 10%-30% smaller.

For $n \geq 512$, the simple shift implementation does not fit in the XC2V3000-4. The simple shift implementation runs in highest frequency for small n, because the circuit is simple and compact for small n. The binary shift implementation uses fewer slices than the simple shift one, but it runs in lower frequency. The binary shift, the left shift, and the right shift implementations are comparable in the number of used slices. The clock frequency of the binary shift is the worst of the four implementations. Recall that the binary shift implementation has two circuits: (1) the Muller-Preparata's adder tree circuit to compute the number of 1's and (2) the multiplexer tree to generate consecutive 1's. Although both circuits has $O(\log n)$ depth, the adder-tree is complicated and has large depth. To confirm this fact from the practical point of view, we have performed the logic synthesis for the Verilog HDL source codes of these circuits



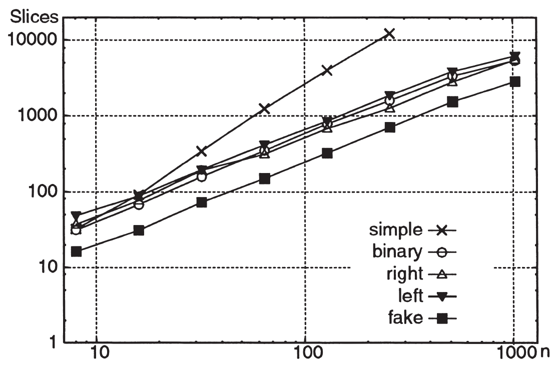


Fig. 2 The performance evaluation for the four implementations and the fake implementation.

independently. The XST logic synthesis tool reported that the adder tree and the multiplexer tree run in 67 MHz and in 244 MHz for $n = 128$. It follows that the adder tree is the bottle neck for the performance of the binary shift implementation. Further, the simple shift implementation runs in 71 MHz for $n = 128$. Hence, as long as the Muller-Preparata's adder tree is used, the clock frequency of the binary shift implementation cannot be better than the simple shift.

The fake implementation in Fig. 2 is an implementation of $C(n, k)$ in which a circuit for computing t is removed. In other words, the fake implementation consists of common circuits of the four implementations. Although the fake implementation does not compute $C(n, k)$ numbers correctly, it is useful to analyze the complexity of the four implementations; It gives the lower bound of the performance of $C(n, k)$ counter implementations in the sense that the performance of any $C(n, k)$ counter implementation cannot be better than that of the fake implementation. For $n = 128$, the fake, the simple shift, and the binary shift implementations runs in 112 MHz, 70 MHz, and 45 MHz, respectively. It follows that, the delay for computing t in the binary shift implementation are dominant when $n = 128$ while that in the simple shift is small. On the other hand, for $n = 128$, the number of used slices are 324, 4034, and 788 respectively. Hence, most of the slices in the simple shift implementation are used to compute t.

Suppose that, for each k $(0 \leq k \leq n)$, a $C(n, k)$ counter is used $h(k)$ times to solve some problem. Recall that the binary shift implementation runs in $\binom{n}{k}$ clock cycles to list all numbers in $C(n, k)$, since it is non-redundant. Thus, it runs in $T = \sum_{0\leq k\leq n} h(k)\binom{n}{k}$ clock cycles if we use the binary shift implementation. On the other hand, the right shift implementation runs approximately in $(1 + \frac{n-k}{n})\binom{n}{k}$ cycles. Since $\frac{n-k}{n} \leq 1$, it runs in no more than

$$\sum_{0\leq k\leq n} h(k)\left(1 + \frac{n-k}{n}\right)\binom{n}{k} \leq 2T$$

clock cycles to solve the problem. Hence, we can guarantee that the right shift implementation (and the left shift implementation) never runs in more than $2T$ cycles. Further, if a $C(n, k)$ counter is used almost symmetrically, that is, if $h(k) \approx h(n - k)$ for all k, it runs in

$$\begin{aligned}&\sum_{0\leq k\leq n} h(k)\left(1 + \frac{n-k}{n}\right)\binom{n}{k}\\ &\approx \frac{1}{2}\sum_{0\leq k\leq n} h(k)\left\{\left(1 + \frac{n-k}{n}\right) + \left(1 + \frac{k}{n}\right)\right\}\binom{n}{k}\Bigg\}\\ &= \frac{3}{2}T\end{aligned}$$

clock cycles. If this is the case, the right shift and the left shift implementations list all numbers faster than the binary shift implementation when their clock frequency is $\frac{3}{2}$ times larger than that of the binary shift. Actually, the right shift and the left shift implementations are faster for $n \geq 256$ and the binary shift implementation is faster for $n \leq 32$ to list all $C(n, k)$ numbers.

8. Applications to the Partial Exhaustive Search

The main purpose of this section is to present how we use a $C(n, k)$ counter for a digital halftoning method presented in [6], which finds a high quality binary image reproducing an original gray-scale image.

Suppose that an original gray-scale image $A = (a_{i,j})$ of size $n \times n$ is given, where $a_{i,j}$ denotes the intensity level at position (i, j) $(1 \leq i, j \leq n)$ taking a real number in the range $[0, 1]$. The goal of halftoning is to find a binary image $B = (b_{i,j})$ of the same size that reproduces original image A, where each $b_{i,j}$ is either 0(black) or 1(white). The halftoning is one of the necessary tasks to print gray-scale images using laser and ink jet printers. We measure the goodness of output binary image B using the Gaussian filter that approximates the characteristic of the human visual system. Let $V = (v_{s,t})$ denote a Gaussian filter, i.e. a 2-dimensional symmetric matrix of size $(2w + 1) \times (2w + 1)$ satisfying $\sum_{-w\leq s,t\leq w} v_{s,t} = 1$, where each $v_{s,t}$ $(-w \leq s, t \leq w)$ is determined by a 2-dimensional Gaussian distribution. The image $C = (c_{i,j})$ restored from a binary image $B = (b_{i,j})$ by applying the Gaussian filter is a gray-scale image:

$$c_{i,j} = \sum_{-w\leq s,t\leq w} v_{s,t} b_{i+s,j+t} \quad (1 \leq i, j \leq n) \qquad (12)$$

From $\sum_{-w\leq s,t\leq w} v_{s,t} = 1$, each $c_{i,j}$ takes a real number in the

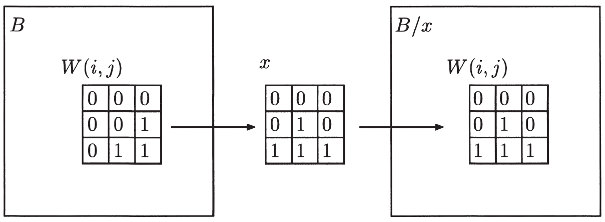


Fig. 3 Illustrating examples of B, $\boldsymbol{x}$, and $B/\boldsymbol{x}$.

Table 3 The values of $\binom{16}{k}$, $h(k)$, and $h(k) \times \binom{16}{k}$.

images		Lena		Plane	
k	$\binom{16}{k}$	$h(k)$	$h(k) \times \binom{16}{k}$	$h(k)$	$h(k) \times \binom{16}{k}$
0	1	955	955	3627	3627
1	16	28111	449776	14290	228640
2	120	129134	15496080	25974	3116880
3	560	253941	142206960	49031	27457360
4	1820	331054	602518280	86867	158097940
5	4368	361361	1578424848	145540	635718720
6	8008	387653	3104325224	197659	1582853272
7	11440	511108	5847075520	224777	2571448880
8	12870	592118	7620558660	214663	2762712810
9	11440	641379	7337375760	204483	2339285520
10	8008	547496	4384347968	242472	1941715776
11	4368	430466	1880275488	492890	2152943520
12	1820	294923	536759860	840587	1529868340
13	560	191399	107183440	941849	527435440
14	120	98839	11860680	655712	78685440
15	16	30038	480608	236462	3783392
16	1	1867	1867	18867	18867
Total	65536	4831842	33169341974	4595750	16315374424

range $[0, 1]$. Thus, the restored image C is an L-level gray-scale image. We can say that a binary image B is a good approximation of original image A if the difference between A and C is very small. According to this consideration, we are going to define the error by the difference between C (or B) and A as follows. The error $e_{i,j}$ at each pixel location (i, j) is defined by

$$e_{i,j} = a_{i,j} - c_{i,j}, \qquad (13)$$

and the total error is defined by

$$Error(A, B) = \sum_{1 \le i,j \le n} |e_{i,j}|. \qquad (14)$$

Since the Gaussian filter approximates the characteristics of the human visual system, we can think that image B reproduces original gray-scale image A if $Error(A, B)$ is small enough.

It is known that a good binary image B with small total error can be obtained by the partial exhaustive search for windows of the binary image [6]. We briefly explain the idea of the partial exhaustive search. Suppose that an original image A and a temporary binary image B are given. Let $W(i, j)$ be a window of size $m \times m$ in B whose top-left corner is at position (i, j), $\boldsymbol{x}$ be a binary pattern in $W(i, j)$, and $B/\boldsymbol{x}$ be the binary image such that $W(i, j)$ of B is replaced by $\boldsymbol{x}$. We refer the reader to Fig. 3 for illustrations of B, $\boldsymbol{x}$, and $B/\boldsymbol{x}$. Let f be a function such that

$$f(\boldsymbol{x}) = Error(A, B/\boldsymbol{x}). \qquad (15)$$

By computing $f(\boldsymbol{x})$ for all possible 2^{m^2} bit patterns $\boldsymbol{x}$, we can obtain an optimal pattern $\boldsymbol{r}$ in formula (2). Clearly, the total error of $B/\boldsymbol{r}$ is not larger than that of B. The idea of the partial exhaustive search is to repeat this operation for a window moving in the raster scan order. A halftoning algorithm presented in [6] first initializes a binary image by random thresholding, and then repeats the partial exhaustive search in raster scan order until no more improvement is possible. The resulting binary images are sharp and high quality, and reproduce the continuous tone of original images very well.

Now, we have the following conjecture in term of formula (15).

Conjecture 3: Function f of formula (15) is concave.

We have no proof for this conjecture, but we believe this conjecture is correct because we cannot find its counter example. Since f satisfies formula (6), we can find the best binary pattern $\boldsymbol{r}$ in formula (2) by the binary search or linear search techniques.

We have performed the linear search technique to find $\boldsymbol{r}$ as follows: Let β be the total intensity in window $W(i, j)$ of a current binary image B, that is,

$$\beta = \sum_{0 \le s,t \le m-1} b_{i+s,j+t}. \qquad (16)$$

Since B is an intermediate solution, it is not "bad" binary

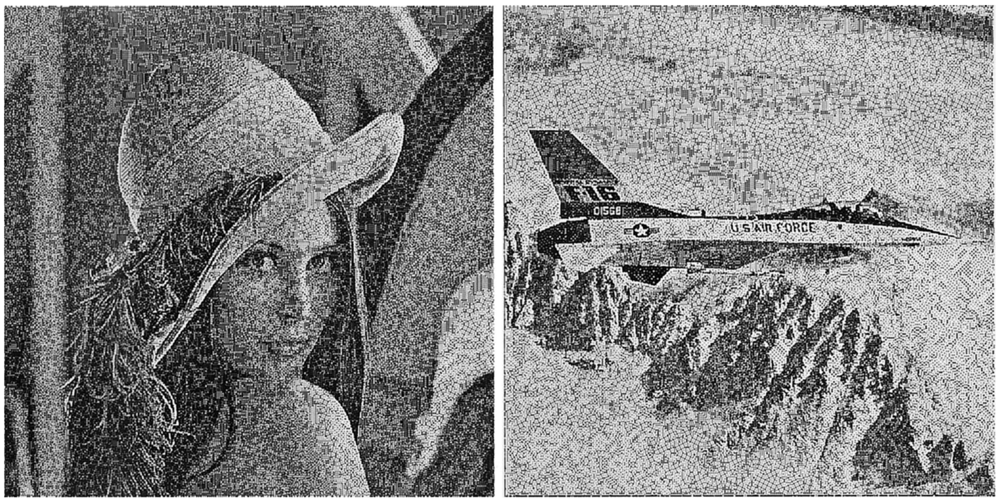


Fig. 4 The resulting image of "Lena" and "Plane."

image. So, the number of 1's in the best binary pattern in $W(i,j)$ must be close to β. Thus, we start the linear search for $f(\boldsymbol{r}_p)$ with $p = \beta$. By increasing and decreasing p in an obvious way, we can find the bottom of the concave sequence $f(\boldsymbol{r}_0), f(\boldsymbol{r}_1), \ldots, f(\boldsymbol{r}_{m^2})$. If we can start with p ($1 \leq p \leq m^2 - 1$) such that $f(\boldsymbol{r}_p)$ is the bottom (i.e. $f(\boldsymbol{r}_{p-1}) > f(\boldsymbol{r}_p) < f(\boldsymbol{r}_{p+1})$), then the linear search just computes $f(\boldsymbol{r}_{p-1}), f(\boldsymbol{r}_p)$, and $f(\boldsymbol{r}_{p+1})$. If we start with $p = 0$ then the linear search may just compute $f(\boldsymbol{r}_0)$ and $f(\boldsymbol{r}_1)$, if $f(\boldsymbol{r}_0)$ is the bottom. These facts allow us to reduce the computing time.

We have developed a digital halftoning system that performs the partial exhaustive search for a window of size 4×4. We have used a PCI-connected board FPGA board [15] with XC2V3000-4. The basic architecture of our hardware implemented in the FPGA is as follows: The host PC stores the original and the current binary image and the FPGA is used as a co-processor. To compute the best binary image $\boldsymbol{r}$ for a window, the host PC sends necessary image data, and the value of k to the FPGA though PCI-bus. The FPGA performs the partial exhaustive search for C$(16,k)$ and returns $\boldsymbol{r}_k$ and $f(\boldsymbol{r}_k)$ to the host PC. By repeating this operation, we obtain a binary image. Since n is small, we use the simple shift implementation of a C$(16,k)$ counter.

Finally, we show experimental results for the 8-bit gray-scale version of well-known standard image "Lena" of size 512×512. Let $h(k)$ ($0 \leq k \leq 16$) denote the number of times $f(\boldsymbol{r}_k)$ is computed until the resulting binary image is obtained. Clearly, the size of the search space to compute $f(\boldsymbol{r}_k)$ is $\binom{16}{k}$ and so the total size of the search space is $h(0)\times\binom{16}{0}+h(1)\times\binom{16}{1}+\cdots+h(16)\times\binom{16}{16}$. The reader should refer to Table 3 for the values of $\binom{16}{k}$, $h(k)$, and $h(k)\times\binom{16}{k}$. The partial exhaustive search is performed for 1598403 times. Hence, to compute $f(\boldsymbol{r})$ for a window, $f(\boldsymbol{r}_k)$ is computed for expected $4831842/1598403 \approx 3.02$ k's. Thus, in most cases, $f(\boldsymbol{r}_k)$ is evaluated only for three k's. Also, $f(\boldsymbol{x})$ is computed for expected $33169341974/1598403 \approx 20751$ $\boldsymbol{x}$'s. Since the simple exhaustive search performs this computation for all $2^{16} = 65536$ $\boldsymbol{x}$'s, we can expect a speedup factor of $65536/20751 \approx 3.15$. Actually, the partial exhaustive search runs in 546 seconds, while the simple exhaustive search runs in 1391 seconds. Thus, the actual speedup factor is $1391/546 \approx 2.55$. Due to the overhead of the PCI bus and local computation, the speedup factor is a bit smaller. Since the circuit has run in 80 MHz, without miscellaneous overhead, it would run in $33169341974/80\,\text{MHz} \approx 415$ seconds. Thus, we have approximately $(546-415)/415 \approx 32\%$ overhead.

We also performed the same experiment for image "Plane". The value of $f(\boldsymbol{r})$ for a window is computed using the partial exhaustive search for 1520546 times to obtain the resulting image. Also, to compute $f(\boldsymbol{r})$, $f(\boldsymbol{r}_k)$ is computed for expected $4595750/1520546 \approx 3.02$ k's and $f(\boldsymbol{x})$ is computed for expected $16315374424/1520546 \approx 10730$ $\boldsymbol{x}$'s. Hence we can expect a speedup factor $65536/10730 \approx 6.11$. The simple exhaustive search runs in 1353 seconds and our partial exhaustive search runs in 335 seconds, and so the actual speedup is $1353/335 \approx 4.04$. Since "Plane" is a whity image, the computation of $f(\boldsymbol{r}_k)$ is performed frequently for large k. We can confirm this fact in Table 3. Thus, we can obtain a higher speedup factor for "Plane". The readers should refer to Fig. 4 for the resulting images. We also note that the simple exhaustive search for "Lena" takes 60,500 seconds [6] on a Pentium 4 based PC (Xeon 2.8 GHz) without using an FPGA.

9. Conclusions

The main contribution of this work is to introduce a C(n,k) counter which lists all n-bit numbers with $(n-k)$ 0's and k 1's, and to present its application to the partial exhaustive search. We have presented several implementations of C(n,k) counters and have evaluated its performance in terms of the number of used slices and the clock frequency for the Xilinx VirtexII FPGA XC2V3000-4. As a real life application we have used a $\mathrm{C}(n,k)$ counter to accelerate a digital halftoning method that generates a binary image reproducing an original gray-scale image. By the partial exhaustive search using a $\mathrm{C}(n,k)$ counter, we accelerated this task and achieved a speedup factor of more than 2.5 over the simple exhaustive search. For a whity gray-scale image, the speedup factor is more than 4.

The partial exhaustive search is helpful for reducing the size of the search space. In particular, if a combinatorial problem is represented by a function f, which is either biased or convex, this approach may work very well. It would be of interest to apply this approach for real life combinatorial optimization problems.

Acknowledgment

The authors would like to thank Dr. Jacir Bordim for his comments to improve the presentation of this paper. Also, they would like to thank anonymous referees for their valuable comments.

References

[1] J.L. Bordim, Y. Ito, and K. Nakano, "Accelerating the CKY parsing using FPGAs," IEICE Trans. Inf. & Syst., vol.E86-D, no.5, pp.803–810, May 2003.

[2] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press, 1990.

[3] A. Gibbons and W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, 1988.

[4] J. Hromkovič, Algorithms for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics, Springer-Verlag, 2001.

[5] Xilinx Inc., Virtex-II Platform FPGAs: Complete Data Sheet, 2003.

[6] Y. Ito and K. Nakano, "FM screening by the local exhaustive search, with hardware acceleration," International Journal of Computer Science, vol.16, no.1, pp.89–104, Feb. 2005.

[7] D.E. Knuth, "Digital halftones by dot diffusion," ACM Trans. Graphics, vol.6-4, pp.245–273, 1987.

[8] D.L. Lau and G.R. Arce, Modern Digital Halftoning, Marcel Dekker, 2001.

[9] R. Lin, K. Nakano, S. Olariu, M.C. Pinotti, J.L. Schwing, and A.Y. Zomaya, "Scalable hardware-algorithms for binary prefix sums," IEEE Trans. Parallel Distrib. Syst., vol.11, no.8, pp.838–850, Aug. 2000.

[10] D.S. Johnson and M.R. Garey, Computers and Intractability: A Guide to the Theory of NP-Completeness, W H Freeman & Co., 1979.

[11] R. Motwani and P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.

[12] D.E. Muller and F.P. Preparata, "Bounds to complexityies of network for sorting and for switching," J. ACM, vol.22, pp.195–201, 1975.

[13] K. Nakano and E. Takamichi, "An image retrieval system using FPGAs," IEICE Trans. Inf. & Syst., vol.E86-D, no.5, pp.811–818, May 2003.

[14] K. Nakano and K. Wada, "Integer summing algorithms on reconfigurable meshes," Theor. Comput. Sci., vol.197, pp.57–77, 1998.

[15] Nallatech, Xtreme DSP Development Kit User Guide, 2002.

[16] B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs, Oxford University Press, 2000.

[17] M. Platzner and G.D. Micheli, "Acceleration of satisfiability algorithms by reconfigurable hardware," Proc. International Conference on Field Programmable Logic and Applications (FPL), pp.69–78, 1988.

[18] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya, "Solving satisfiability problems using reconfigurable computing," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.9, no.1, pp.109–116, Feb. 2001.

[19] P. Zhong, P. Ashar, S. Malik, and M. Martonosi, "Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with boolean satisfiability," Design Automation Conference, pp.194–199, 1998.

Koji Nakano received the BE, ME and Ph.D degrees from Osaka University, Japan in 1987, 1989, and 1992 respectively. He had worked at Hitachi Ltd, the Department of Electrical and Computer Engineering, Nagoya Institute of Technology, and the School of Information Science, JAIST. He is currently a full professor with the School of Engineering, Hiroshima University. He has published papers in IEEE Transactions on Parallel and Distributed Systems, Journal of Parallel and Distributed Computing, Theoretical Computer Science, Theory of Computing Systems, Parallel Algorithms and Applications, and IEICE Transactions. He has served as an editor of IEEE Transactions on Parallel and Distributed Systems, IEICE Transactions on Information and Systems, and International Journal of Foundations on Computer Science. He has served on the program chair and committee member of conferences and workshops including International Conference on Parallel Processing, International Parallel and Distributed Processing Symposium, Workshop on Advances in Parallel and Distributed Computational Models, Workshop on Wireless Networks and Mobile Computing, and Reconfigurable Architecture Workshop. His research interests includes mobile computing, hardware algorithms, reconfigurable computing, parallel algorithms and architectures.

Youhei Yamagishi received his B.S. degree in physics from Konan University, and M.E. in information science from Japan Advanced Institute of Science and Technology (JAIST) in 2001 and 2003. He is currently on Hiroshima University as a Ph.D. candidate. His research interests include image processing.