Design of A Dynamic Pipelined Architecture for Fuzzy Color Correction

Jer-Min Jou, Shiann-Rong Kuang, Yeu-Horng Shiau, and Ren-Der Chen

***Abstract*—Color correction, which nonlinearly converts the color coordinates of an input device such as the scanner and digital camera into that of an output device such as the color laser printer, is important for multimedia applications. In this brief, we present a novel dynamic pipelined VLSI architecture for the fuzzy color correction algorithm (FCC) proposed by Jou *et al.* to meet the speed requirement of time-critical applications. To promote the performance, the presented architecture is dynamically pipelined with unfixed or run-time determined latencies (or data initiation intervals) and the speculation technique is also applied, then the problems of arduous pipelining, due to the variant execution time of each iteration and slower executing of FCC are solved efficiently. As for data path design, a systematic design methodology of high-level synthesis is used. As a result, a significant (about 2 times) speedup of the dynamic pipelined architecture with a slight hardware overhead relative to the sequential one has been achieved.**

***Index Terms*—Color correction, dynamic pipeline, fuzzy.**

I. Introduction

The reproduction of color documents between different devices has become one of important problems in multimedia applications due to the rapid expansion on the using of color input/output devices such as color scanners, digital cameras, and color printers. The major issue of a color correction system is how to maintain the original color quality of documents or images very faithfully when they are transferred between different input/output devices. The most common example is the color documents scanned in by the scanner and then printed out by the printer, it is also this brief's focusing problem. Since the tints and scales of colors between different devices are different, the coordinates obtained by the color scanner cannot be used directly by the color printer and then the color of the reproduced document after transferring may be distorted very much and be unacceptable. In additional, the nonlinear color-mapping problem makes the color correction process difficult and slow. Therefore, a mechanism to efficiently (smoothly) and fast convert the scanner's color coordinates into that of the printer is necessary and important for a time-critical color office automation environment.

In the past, there were several methods [2]–[5] proposed to deal with the color correction (or conversion) problem between the scanner and the printer. Some of them are with high computation costs and/or need large storage area [2], [3]; the others are slow and unsuitable for time-critical applications [4], [5]. Recently, a good fuzzy color correction algorithm and a corresponding FPGA implementation were presented in [6], however, they are slow and even with poor performance due to unreduced computations and the maximum criteria defuzzification used. In [1], an efficient fuzzy-tree correction algorithm, denoted as FCC, is proposed. Using fuzzy inference trees like [6], FCC converts the color coordinates of red, green, and blue (RGB) generated by the scanner to that of cyan, magenta, and yellow (CMY) of the color printer smoothly. The advantages of FCC lie in its simplicity, adaptability and good correction effect. A sequential hardware of it had also been designed [1], but its processing speed is still slow and is a problem for time-critical applications. Making a fast hardware realization for FCC to promote the color correction speed is then the subject of this brief.

We present here a dynamic pipelined architecture for FCC to increase the correction speed significantly. Pipelining [7]–[10] is a powerful and popular technique in designing high throughput digital circuits. For a functional pipeline design, two consecutive iterations of the same loop are initiated at a time interval called the latency. In general, the latency of a pipeline is fixed [7]–[9] or has some fixed values [10]. However, in the main FCC processing loop, variant execution time of each iteration and time-relative data dependencies between different iterations make pipeline latencies unfixed and pipelining be hard to do. In order to construct a high throughput pipelined architecture for FCC, a new dynamic pipelined architecture with run-time determined latencies will be designed. We first identify and modify some complicated data and/or control dependencies in FCC such that pipelining is easy to carry out and then we partition FCC into two sections: an inner section and a main section. In the main section, the whole inner section is viewed as an unbound delay operation whose delay is data-dependent and run-time determined. The inner and main sections are then optimized in performance by using speculation-based pipelining [11] to form a single dynamic pipeline. An integrated controller is designed to control the dynamic pipelined datapath with variant latencies. After dynamic pipelining, the pipeline latencies of the FCC loop are dependent on the execution length of the inner section and are naturally unfixed and vary between 5 and 12. The average value is about 9.5 (measured by the hardware simulation). Thus, the processing speed of FCC can be promoted significantly and is about 2 times speedup to the sequential version [1], but hardware overhead is slight.

II. Overview of FCC and Its Sequential Architecture

In [6], a fuzzy algorithm with good color correction performance was proposed. However, it has high computation complexity and tends to be slow. Based on [6], we had developed a more efficient FCC algorithm [1], which uses a new efficient approach for fuzzy inference and the different center-average method for defuzzification and gets the same color correction performance as the algorithm of [6]. During the fuzzy color correction process of [1], the function of each fuzzy subtree is to do one color conversion that is performed by finding a decision path in it. At the beginning, a subtree with one level and eight leaves is employed to determine the mapping of the red color according to the matching degree of the input, each input color value X_i is processed with eight fuzzy sets $(s_1, s_2, \ldots, s_8)$ corresponding to the eight leaves for the color. The eight fuzzy sets are used to represent the different intensities of each color. After determining the decision path in the tree for the red color, the subtrees and decision paths of green and blue colors can be found sequentially on the analogy of the red color fuzzy inference process. The FCC algorithm of [1] is presented in Fig. 1. In it, L denotes the current level of the three-level (color) fuzzy tree and $1 \leq L \leq 3$. If $L = 1$, then X_i is red and if $L = 2$, then X_i is green; otherwise, X_i is blue. In addition, $Path_L$ denotes the decision path of the current subtree and $0 \leq Path_L \leq 7$ and a 146 bytes ROM is used as the rule memory. The s^1 (the supported value of fuzzy set s_1) and the d (the fixed distance between any two neighbor fuzzy sets of each subtree and it is a constant) of the blue color subtree are stored in the first 128 bytes of the ROM and the s^1 and the d of green and red

Manuscript received September 15, 2000; revised July 29, 2001. This work was supported in part by the National Science Council, Republic of China, by Grant NSC-86-2221-E-006-022. The proposed systematic approach to design a dynamic pipelined architecture has taken out the USA and R.O.C. patents.

J.-M. Jou, Y.-H. Shiau, and R.-D. Chen are with the Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan, R.O.C. (e-mail: jou@j92a21.ee.ncku.edu.tw).

S.-R. Kuang is with the Department of Computer Science and Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C. (e-mail: srkuang@cse.nsysu.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2002.808458

```
L=1;
while (input pattern X_i ≠ NULL) {
        Calculate the address of rule memory (ROM);
        s^l = ROM[address++];   D = X_i − s^l;
        k = 0;    Path_L = 0;    d = ROM[address];
        while (k < 8 && D > 0) {
                D = D − d;   Path_L = k;   k++;
        }
        if (1 ≤ k ≤ 7 && |D| ≤ d/2) Path_L = k;
        Calculate X_o using (2);
        if (++L == 4) L = 1;
}
```

Fig. 1. The FCC algorithm [1].

color subtrees are stored in the following locations. The address for retrieving the s^1 and the d from ROM for the corresponding tree level are calculated as

$$\text{address} = \begin{cases} 144, & \text{if } L = 1 \\ 128 + Path^*_{L-1}2, & \text{if } L = 2 \\ Path^*_{L-2}16 + Path^*_{L-1}2, & \text{otherwise} \end{cases}. \quad (1)$$

The center-average based inference result X_o is calculated by

$$X_o = \begin{cases} w_1, & \text{if } k = 0 \\ \frac{|D|*w_k + (d-|D|)*w_{k+1}}{d}, & \text{if } 1 \le k \le 7 \\ w_8, & \text{if } k = 8 \end{cases} \quad (2)$$

where $|D|$ is the distance between X_i and the fuzzy set s_k and $w_1, w_2,\ldots$ and w_8 are the supported values of the fuzzy sets which represent the various intensities of color space CMY. For more details of the fuzzy color correction process and performance, please refer to [1].

Fig. 2 shows the corresponding sequential architecture of FCC in [1]. In Fig. 2, the ALU performs multiplication and division operations in (2). The notation $\ll n (\gg n)$ denotes that the value is shifted left (right) n bits. The storage used to store k is a 4-bit up-count counter to perform its increment operation. The condition $E8$ ($E0$) is set to 1 if $k = 8$ ($k = 0$). Another condition $L0$ is set to 1 if $D > 0$. Moreover, $w_1, w_2,\ldots$ and w_8 are stored in an 8-byte ROM, denoted as ROM2 and k is used as its address.

Note that the execution time of each iteration in the FCC main loop section is variable, since the number of its inner loop's iterations (see the second while loop of Fig. 1) is changed according to the different values of data k and D. In addition, FCC also contains data dependent branches, which can not be sped without speculative computation which makes tasks executing before this is known to be necessary whenever no other task is ready for execution [11]. Thus, pipelining FCC with traditional pipeline methods that use fixed latencies to design pipelined chips is impossible (that is why only a sequential circuit for FCC is presented in [1]) and a new dynamic pipelined architecture with variant latencies combined the speculative technique is required for designing a higher performance FCC circuit.

III. Dynamic Pipelined Architecture Design of FCC

To efficiently design the dynamic pipelined architecture, FCC first is modeled as a hardware behavioral description and then is partitioned into the inner section and the main section. Next, the inner and main sections are optimized by using speculative computation and pipelining to enhance the performance. Finally, the performance-optimized inner and main sections are combined to form a dynamic pipelined datapath with variant latencies and then the dynamic pipelined controller is generated. The following subsections explain these processes.

A. Graph Model and Partition

To clearly explain the dynamic pipelined architecture, the FCC description in Fig. 1 is transformed into a hardware behavioral description and then is represented as a simplified control/data flow graph (CDFG) shown in Figs. 3 and 4, respectively. In the simplified CDFG of Fig. 4, only the important operations and dependencies are depicted. In it, the notations of operations addition, subtraction, multiplication, division, comparison, up-count and assignment are denoted as $+, -, *, /, \diamondsuit, ++$ and $\leftarrow$, respectively. Moreover, the operation of Fig. 4 also in line i of Fig. 3 is denoted as notation-i. For example, the addition operation in line 4 of Fig. 3 is denoted as $+4$ as shown in Fig. 4. The multiplication and division operations need two clock cycles to compute the values and the other operations like addition, subtraction, or comparison need one cycle to execute based on the 0.8 μm cell library. The control condition produced by the comparison operation in line i of Fig. 3 is denoted as ci. In Fig. 3, if condition $c2$ ($c9$) is true, the main (inner) while loop will continue to execute; otherwise, the loop will terminate. There is an inner loop with data dependent number of iterations in the main loop of the simplified CDFG. Pipelining this CDFG with a fixed latency is impossible due to the variant-iteration (variant execution-time) inner loop.

To design a high performance dynamic pipelined FCC architecture with minimum hardware resource, all operations of Fig. 3 are first partitioned into two parts: a main section and an inner section. The operations of the second while loop in lines 9 and 10 as well as the operations in line 11 of Fig. 3, which make the execution length of FCC main loop to be variable and be unpredictable in advance, are grouped into the inner section and the remaining operations are grouped into the main section. The operations in line 11 are put into the inner section too since line 10 and line 11 both have the common operation: $Path_L = k$. The aim of the partition is to divide the unpredictable inner section from the main section so that they can be pipelined to form a high performance dynamic pipeline. On the other hand, although the operations in line 12 to line 17 of Fig. 3 also make the execution time of each iteration of the main loop section to be variable, but its influence is predictable. We know that if k is equal to 0 or 8 then only operation $\leftarrow$ 12 is executed; otherwise operations in line 13 to line 17 are executed. That is, we can know its execution time needs 1 or 7 clock(s) in advance. Therefore, the operations in line 12 to line 17 are not put into the inner section.

B. Speculative Computation of Inner Section

After the CDFG for FCC is partitioned, the inner section is scheduled and pipelined by applying performance optimization techniques to achieve a high performance. The inner section is performance-optimizedly scheduled without regard to the interaction and precedence constraints between it and the main section temporarily. The operations of the inner section may be scheduled into 4 states by using the conventional pipeline scheduling [12]: the operations in line 9 of Fig. 3 are scheduled at the first state; the operations in line 10 are scheduled at the second state; and the operations in line 11 are scheduled at the third and fourth states, respectively. However, this schedule is slow and cannot be pipelined due to data dependency between operations $\diamondsuit 9$ (i.e., $k < 8$) and $++10$ (i.e., $k++$) (see Fig. 3). To get a high performance circuit before pipeline scheduling, the technique of control speculative [11], which implies the execution of an operation before the execution of a preceding instruction on which it is control dependent, must be applied.

We first schedule the comparison operations $\diamondsuit 9$ in line 9 and the conditional operations in line 10 into a certain state called S_α and then determine whether or not the results generated by the operations in

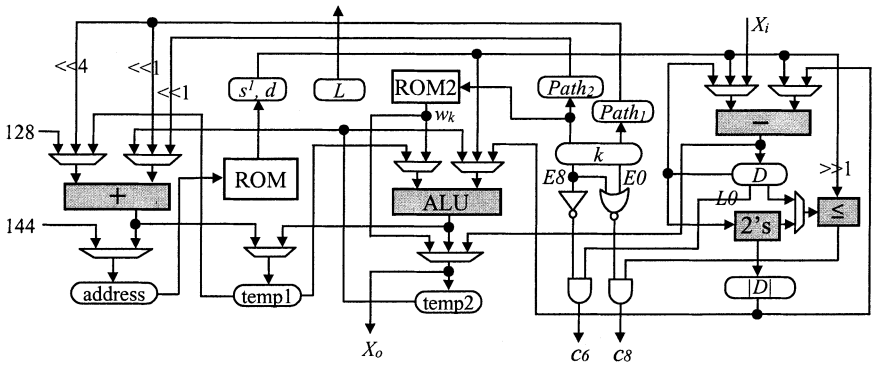

Fig. 2. The sequential architecture of FCC [1].

```
1:    L=1;
2:    while (input pattern Xi ≠ NULL) {
3:        if (L==1) address=144;
4:        if (L==2) address=128+ Path1*2;
5:        else address=Path1*16+Path2*2;
6:        s'=ROM[address++];
7:        D= Xi − s' ;    k=0;    PathL=0;
8:        d=ROM[address];
9:        while (k<8 && D>0) {
10:           D= D − d;   PathL=k;   k++;
          }
11:       if (1 ≤ k ≤ 7 && |D| ≤ d/2)   PathL=k;
12:       if (k==0 || k==8)   Xo = wk;
          else {
13:           t1=|D|*wk;
14:           t2= d − |D|;
15:           t3=t2*wk+1;
16:           t4=t1+t3;
17:           Xo=t4/d;
          }
18:       if (++L==4) L=1;
      }
```

Fig. 3. The hardware behavioral description of FCC.

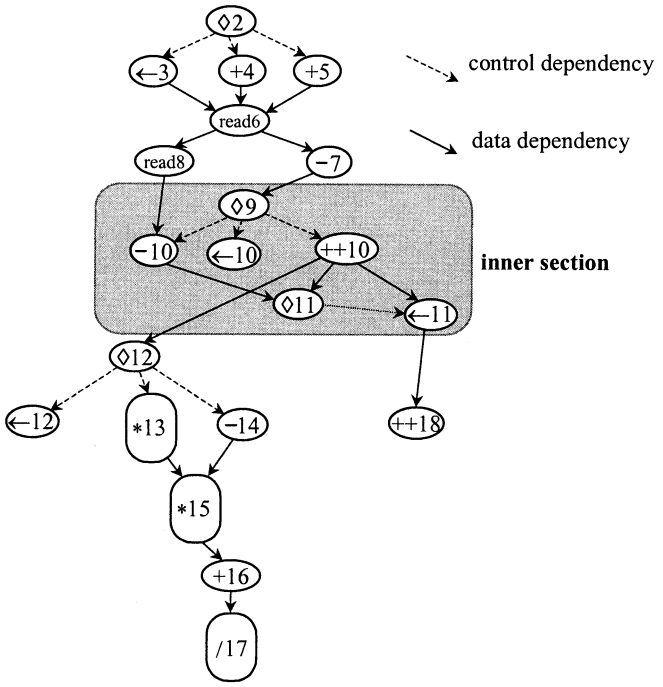

Fig. 4. The simplified CDFG of FCC.

line 10 are stored into registers according to the control condition $c9$ at the ending of state S_α. That is all operations in the inner section are scheduled into one state S_α. However, at state S_α, how to prevent these conditional operations in line 10, whose executions depend on the result of logical operation ◊9, overrunning (when $c9$ is false) becomes a pressing problem. Some new control signals must be planned and associated circuits must be designed to solve it. Let L_D, L_Path_1, L_Path_2 and C_k be the original loading or up-counting control signals of registers D, $Path_1$, $Path_2$ and counter k, respectively. Signals L_D, L_Path_1, L_Path_2 and C_k are set to 1 by the controller when the FCC architecture is in state S_α. To prevent the speculative overruns occurring, some new signals must be designed to replace them to control respective registers or counters. Let their corresponding new control signals be L_D', L_Path_1', L_Path_2' and C_k', respectively, and signal In_α be a control signal which indicates whether the FCC architecture is at state S_α and is set to 1 if it is at state S_α; otherwise, signal In_α is set to 0. Then, signal L_D' is generated by

$$L\_D' = \begin{cases} L\_D, & \text{if } In_\alpha = 0 \\ c9, & \text{otherwise.} \end{cases} \tag{3}$$

Signal C_k' is generated by

$$C\_k' = \begin{cases} C\_k, & \text{if } In_\alpha = 0 \\ c9, & \text{otherwise.} \end{cases} \tag{4}$$

Signal L_Path_1' and L_Path_2' are generated by

$$L\_Path_L' = \begin{cases} L\_Path_L, & \text{if } In_\alpha = 0; \\ 1, & \text{if } In_\alpha = 1 \text{ and } (c9 = 1 \text{ or } c11 = 1) \\ 0, & \text{if } In_\alpha = 1 \text{ and } c9 = 0 \text{ and } c11 = 0 \end{cases} \tag{5}$$

where subscript $_L$ is equal to 1 or 2. Then, the circuits to implement (3) and (5) are shown in Fig. 5(a) and (b), respectively. The circuit to implement (4) is similar to the circuit in Fig. 5(a). By the design above, all operations of the inner section can be scheduled speculatively and efficiently into the state S_α, which further enhances the performance of the FCC architecture and needs only slight additional hardware.

C. Pipelining Main Section

Subsequently, the main section will be pipelined and then integrated with the performance-optimized inner section to form a dynamic pipelined design. Performance optimization of the main section is

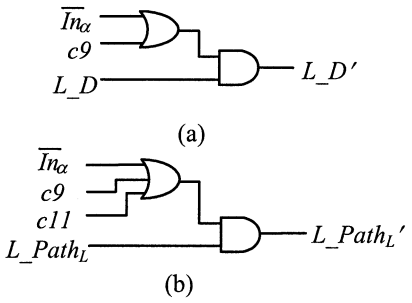

Fig. 5. The circuits for generating signals L_D' and L_Path_L'.

more important and difficult, because the more complex interactive precedence relations between respective sections must be considered and the execution length of the inner section is unknown (data dependent) and unpredictable. We incrementally unwind the main section loop to pipeline the main section. After a polynomial (and in practice small) number of iterations have been unwound and parallelized, a repeating pipeline body will provably emerge.

During pipelining the main section, the original location of the operations of the inner section in the main section is replaced by an unbound delay operation whose delay is dependent on the result of comparison operation $\diamondsuit 9$. Fig. 6 shows the four times unwinding of the main loop body. These operations of calculating the ROM address (e.g., operations $+4$ and $+5$) in each iteration must be scheduled after the operation $+ +\ 18$ of its previous iteration due to the data dependency caused by L. Therefore, the performance of pipeline scheduling shown in Fig. 6 is limited by the data dependency and is highest under the constraint. Note that operations $+4$ and $+5$ are mutually exclusive and thus they can share the same adder though they are scheduled at the same clock cycle. Consequently the scheduling is area-efficiency and is with the minimal hardware resource. In addition, the operation $\leftarrow 12$ and operations ^{*13}, -14, ^{*15}, $+16$, and $/17$ are also mutually exclusive. Thus the condition $c12$ must be appropriately preserved during pipelining to judge these conditional operations whether or not are executed.

After the repeating pipeline body of the main section is found, all operations of different iterations or different sections which are executed at the same time are formed a state such as PS_1 to PS_4 as shown in Fig. 6. The latency ℓ of the pipeline scheduling is equal to four clock cycles plus the delay of the inner section (i.e., the unbound delay operation) and is variable. Since the number of iterations, denoted as I, of the inner section in Fig. 3 is between 1 to 8 and all operations in the inner section are executed within the state S_α. The execution length of the unbound delay operation is also between from 1 to 8. As a result, the latency $\ell = 4 + I$ and its values are between 5 to 12.

In the pipeline scheduling of Fig. 6, no operation of the main section is scheduled at the same state with the unbound delay operation. In fact, operations $+16$ and $/17$ can be scheduled one clock cycle forward to decrease one clock cycle time in executing the main section once; the new pipeline scheduling doesn't increase extra functional units. However, we do not adopt the new pipeline scheduling based on two reasons. First, the performance improvement of this new pipeline scheduling is very little. Second, the new pipeline scheduling swells more design complexity. Therefore, we adopt the original pipeline scheduling.

D. Dynamic Pipelined Datapath

Finally, the state transition graphs of the inner and main sections are generated and combined into a final state transition graph (STG) as shown in Fig. 7. The STG consists of the prelude, repeating pipeline body and postlude. The operations activated in each state of Fig. 7 are not shown and please refer to Fig. 6. The datapath allocation is then

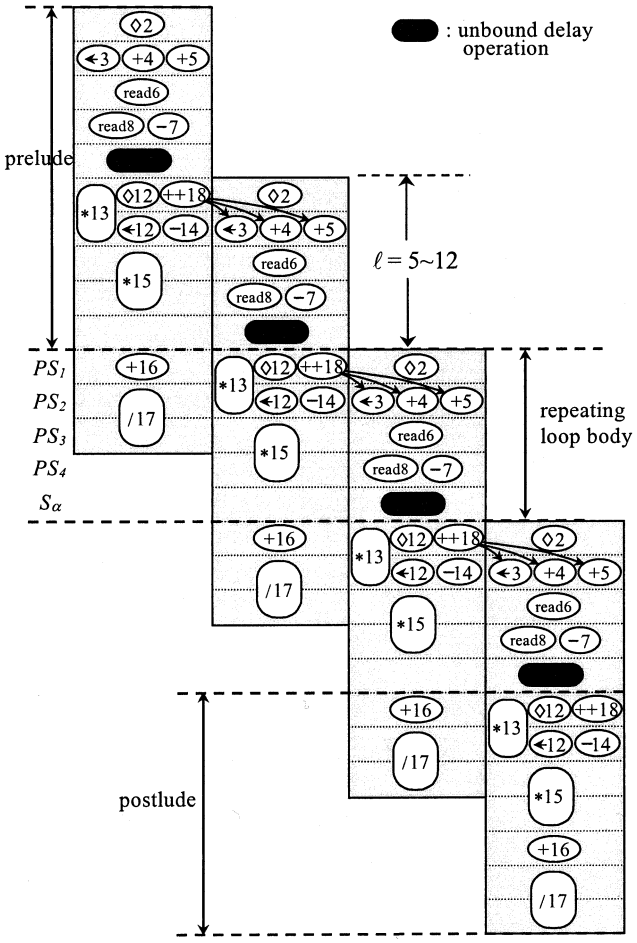

Fig. 6. Result of unwinding of the main section four times.

performed to construct the dynamic pipelined datapath. The datapath allocation technique in [12] is applied to generate the datapath by assigning each operation to a functional unit, by assigning each data value to a storage element and by providing interconnections among functional units and storage elements using multiplexers and/or buses.

The activated times of operations in the repeating loop body can be observed from Fig. 6. By the result, the ALU of Fig. 2 is split into a multiplier and a divider due to their activated time conflict. In addition, operations $+4$ and $+5$ are mutually exclusive and they can share the same adder as mentioned above. The functional units allocated include one adder, one subtractor, one multiplier, one divider, one comparator and one two's complementor. On the other hand, the lifetimes of data values in the repeating pipeline body are shown in Fig. 8, which shows the lifetime of data values in the iteration 1, 2, and 3 of the repeating pipeline body and data value x of the ith iteration in it is denoted as $x(i)$. By Fig. 8, the results of register allocation are shown in Table I. After the functional units and registers have been allocated, the interconnection among them is constructed by tracing the execution of the operations in Figs. 6 and 7. The final dynamic pipelined datapath then is completed and depicted in Fig. 9. A controller is designed to sequence datapath hardware units with run-time determined latencies according to dynamic pipelined scheduling, which is simplified represented by with the state transition graph shown in Fig. 7.

IV. Experimental Results

The dynamic pipelined FCC architecture aforementioned has been designed and simulated in Verilog based on the 0.8-μm cell library.

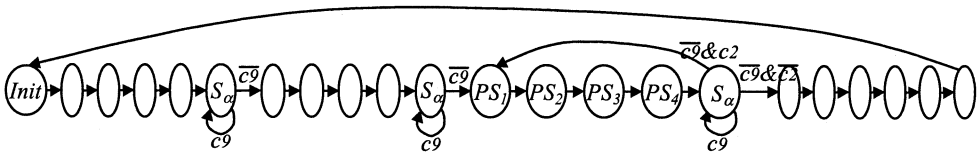

Fig. 7. The final state transition graph.

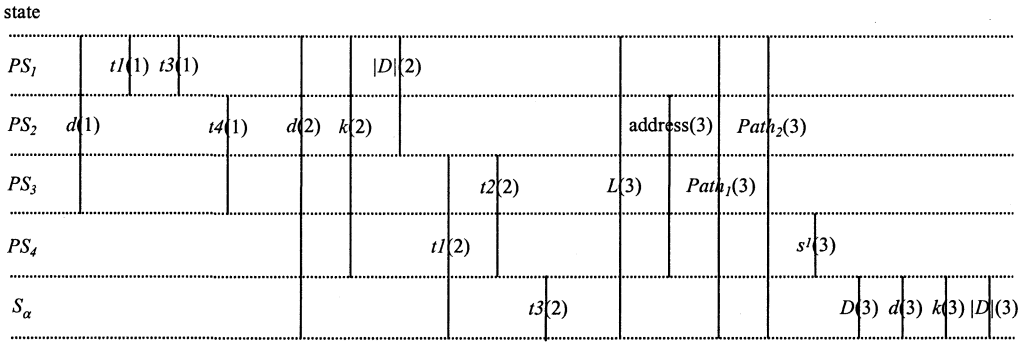

Fig. 8. The lifetimes of data values in the repeating pipeline body.

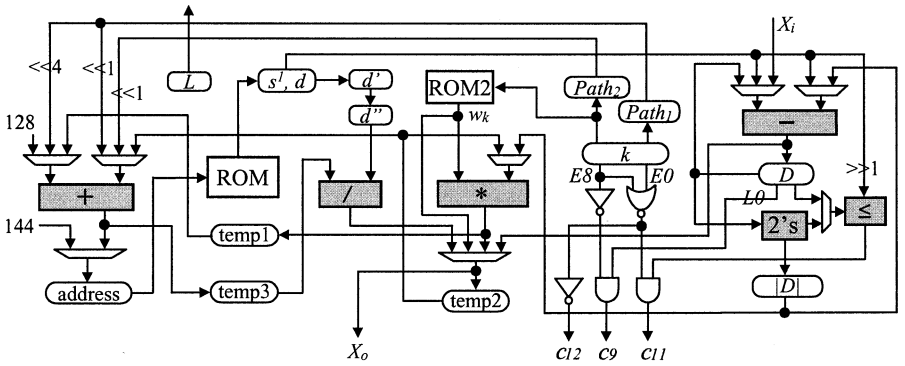

Fig. 9. The dynamic pipelined architecture of FCC.

TABLE I
REGISTER ALLOCATION OF THE DYNAMIC PIPELINING DESIGN

state	register												
	address	L	d	d'	d''	temp1	temp2	temp3	$Path_1$	$Path_2$	k	D	$\|D\|$
PS_1		L(3)		d(2)	d(1)	$t1$(1)	$t3$(1)		$Path_1$(3)	$Path_2$(3)	k(2)		$\|D\|$(2)
PS_2	address(3)	L(3)		d(2)	d(1)			$t4$(1)	$Path_1$(3)	$Path_2$(3)	k(2)		$\|D\|$(2)
PS_3	address(3)	L(3)		d(2)	d(1)	$t1$(2)	$t2$(2)	$t4$(1)	$Path_1$(3)	$Path_2$(3)	k(2)		
PS_4	address(3)	L(3)	s^1(3)	d(2)		$t1$(2)	$t2$(2)		$Path_1$(3)	$Path_2$(3)	k(2)		
S_α		L(3)	d(3)	d(2)		$t1$(2)	$t3$(2)		$Path_1$(3)	$Path_2$(3)	k(3)	D(3)	$\|D\|$(3)

Four pictures were used to test the processing speed of the dynamic pipelined design and the two sequential designs [1], [6]. The compared results with the two sequential designs are reported in Table II. In it, column "file size" shows the file sizes of these pictures. Moreover, columns "sequential [6]," "sequential [1]," and "dynamic pipelining [*]" show the total state number required for two sequential designs and the proposed dynamic pipelined design to process these pictures, respectively. The results show that averagely about 2.7 and 2 times speedup compared with the sequential designs, respectively, can be obtained.

Besides the state numbers shown in Table II, the simulation results show that the latency values of the dynamic pipelined deign vary from 5 to 12. The average latencies $\bar{L}$ for the dynamic pipelined design to process the four pictures are also listed in Table II. From the point of theoretical analysis, we can find that from Fig. 6 the execution time of each iteration of the sequential FCC architecture [1] needs 19 cycles, since it shall use the longest 12 cycles to process the inner section. But the proposed dynamic pipelined FCC architecture only needs 9.5 cycles on average, which is the average value of $\bar{L}$s of the four cases shown in Table II, to run each iteration of FCC. Therefore, we get the

TABLE II
COMPARING RESULTS OF THE THREE FCCS

pictures	file size (bytes)	sequential [6]	sequential [1]	dynamic pipelining [*]	$\bar{L}$	speedup [6]/[*]	speedup [1]/[*]
Pic1	148416	3556800	2576595	1307916	9.25	2.72	1.97
Pic2	230604	5068800	4168651	2148789	10.39	2.36	1.94
Pic3	974916	23392800	16764288	8059754	8.35	2.90	2.08
Pic4	1137198	27287568	19138743	9764665	9.11	2.79	1.96

theoretical speedup equal to $19/9.5\ (=2)$, which is almost the same the experimental values shown in Table II.

However, hardware overhead is needed to achieve the 2 times speedup. Comparing the sequential and dynamic pipelined datapaths shown in Figs. 2 and 9, the penalty paid for the speedup is extra area overhead including more pipelined registers and a somewhat larger controller. In addition, the total area of one multiplier and one divider used in the dynamic pipelined architecture is larger than the area of one ALU in the sequential architecture that can perform multiplication and division operations. In the future, how to reduce the area of datapath such as using the table look-up method to replace some larger functional units is our next goal.

V. CONCLUSION

This brief has presented a dynamic pipelined architecture for the fuzzy tree color correction algorithm FCC. It describes the design process of the FCC dynamic pipelined architecture, which performs a nested loop with data-dependent number of iterations and data-dependent branches, using the variable latencies and speculative computation. The pipeline latencies of the designed architecture are depended on the execution length of the inner section and are naturally unfixed. Experimental results show that the dynamic pipelined FCC architecture obtains about 2 times speedup with a slight area overhead.

REFERENCES

[1] J.-M. Jou, S.-R. Kuang, and R.-D. Chen, "A new efficient fuzzy algorithm for color correction," *IEEE Trans. Circuits Syst. I*, vol. 46, pp. 773–775, June 1998.
[2] R. Takeuchi, M. Tsumura, M. Tadauchi, and H. Shio, "Color image scanner with an RGB linear image sensor," *J. Image Technol.*, vol. 14, pp. 68–72, 1988.
[3] P. C. Hung, "Colorimetric calibration in electronic imaging devices using a look-up-table model and interpolations," *J. Electron. Imag.*, vol. 2, pp. 53–61, 1993.
[4] H. R. Kang and P. G. Anderson, "Neural network application to the color scanner and printer calibrations," *J. Electron. Imag.*, vol. 1, pp. 125–135, 1992.
[5] C. W. Chang and P. R. Chang, "Neural plant inverse control approach to color error reduction for scanner and printer," in *Proc. Int. Conf. Neural Networks*, 1993, pp. 1979–1983.
[6] B. D. Liu and C. Y. Huang, "Design and implementation of the tree-based fuzzy logic controller," *IEEE Trans. Syst., Man Cybern. B: Cybern.*, vol. 27, pp. 475–487, June 1997.
[7] E. M. Circzyc, "Loop winding—A data flow approach to functional pipelining," in *Proc. Int. Symp. Circuits Syst.*, May 1987, pp. 382–385.
[8] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 356–370, Mar. 1988.
[9] L. F. Chao, A. LaPaugh, and E. H.-M. Sha, "Rotation scheduling: A loop pipelining algorithm," in *Proc. ACM/IEEE Design Automation Conf.*, 1993, pp. 566–572.
[10] H. S. Jun and S. Y. Hwang, "Automatic synthesis of dynamically configured pipelines supporting variable data initiation intervals," *IEEE Trans. VLSI Syst.*, vol. 4, pp. 279–285, July 1996.
[11] U. Holtmann and R. Ernst, "Experiments with low-level speculative computation based on multiple branch prediction," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 262–267, 1993.
[12] J.-M. Jou, C.-C. Chin, and Y.-R. Li , "PASS: A package for automatic scheduling and sharing pipelined data paths," in *Proc. Int. Symp. Circuits Syst.*, 1991, pp. 1769–1772.

Theoretical Analysis of Bus-Invert Coding

Rung-Bin Lin and Chi-Ming Tsai

***Abstract*—The closed-form formulas derived from the Markov chains depicting the bus-invert coding processes for even and odd bus widths are employed to compute switching activity and weight per bus transfer, to explain the diminishing returns with increasing bus width and to show the nonviability of partitioning a bus into smaller buses of odd number bits, etc. Probing into the codewords generated by the coding process, we obtain the same closed-form formulas as that obtained by the Markov-based approach. Our contributions are forming a superset over the previously published theoretical results and providing comprehensive background information for exploiting the bus-invert method for low-power applications.**

***Index Terms*—Analysis, low-power design, switching activity, system-level, VLSI.**

I. INTRODUCTION

The power consumption of an off-chip bus includes the power consumed by the drivers, by the capacitance loads on the bus lines and for a parallel terminated bus, by the static currents flowing through the pull-up resistors. Because of the large driver size, heavy interconnect load and requirement for operating in a higher supply voltage, an off-chip driver can consume up to 90 mW [1]. Other studies [2], [3] have also shown that off-chip buses dissipate a substantial amount of the total power. Hence, bus-coding methods that reduce the switching activity or the number of 1s on a bus have been proposed for applications where the power consumption has the utmost importance. Because of the cost, flexibility, time-to-market, technological limitations,

Manuscript received November 7, 2000, revised May 12, 2002. This work was supported in part by the National Science Council, Taiwan, under Grant NSC 89-2215-E-155-010.

The authors are with the Department of Computer Engineering and Science, Yuan Ze University, Chung-Li, 320 Taiwan, R.O.C. (e-mail: csrlin@cs.yzu.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2002.808456