01 Jan 2000

A Systolic Image Difference Algorithm for RLE-Compressed Images

Fikret Erçal
Missouri University of Science and Technology, ercal@mst.edu

Mark Allen

Hao Feng

A Systolic Image Difference Algorithm for RLE-Compressed Images

Fikret Ercal, *Senior Member*, *IEEE*, Mark Allen, and Hao Feng

Abstract—A new systolic algorithm which computes image differences in run-length encoded (RLE) format is described. The binary image difference operation is commonly used in many image processing applications including automated inspection systems, character recognition, fingerprint analysis, and motion detection. The efficiency of these operations can be improved significantly with the availability of a fast systolic system that computes the image difference as described in this paper. It is shown that for images with a high similarity measure, the time complexity of the systolic algorithm is small and, in some cases, constant with respect to the image size. A formal proof of correctness for the algorithm is also given.

Index Terms—Systolic algorithm, image difference, image compression, run-length encoding.

1 Introduction

Binary image processing is used in many areas including robot vision and industrial inspection [1], [2], character recognition, fingerprint analysis, motion detection for safety and security [3], [4], feature extraction [5], map analysis [6], etc. It is a common practice to build special purpose hardware to process binary images in real-time. There are numerous proposals and implementations of such operations in hardware including convolution [7], template matching, component labeling [8], morphological operations, min/max filtering [9], thinning [10], etc. To speed up the process, most hardware approaches utilize pipelining [1], array processors, or systolic architectures [7], [8], [9], [10].

While there are software approaches to processing binary images in compressed form (e.g., run-length encoding (RLE)) to save time and space, hardware approaches rarely operate in compressed mode. To the best of our knowledge, there are no hardware implementations of fundamental image operations which process images in compressed mode without decompressing them. Combined with the power of the hardware, this approach is expected to result in significant performance increases. In this study, we describe a systolic architecture to process binary images in compressed form.

One of the areas where such a system would have significant impact is the inspection of printed circuit boards (PCBs). This work is mainly motivated by the need to speed up the PCB inspection process [2]. Online automatic inspection of PCBs requires acquisition and processing of gigabytes of binary image data in a matter of seconds. Most PCB inspection systems use a reference based approach which requires comparison of the board image against the original CAD design. Therefore, the binary image difference operation is a fundamental step in the inspection process and the system performance critically depends on the speed of this operation. To increase the performance further, run-length encoding (RLE) is used for storage and operations.

Systolic systems use cellular iterative computations and perform global tasks through exchange of local data in a pipelined fashion [11]. Since most of the image processing operations exhibit high local dependencies among data elements, systolic machines are widely used in image processing applications such as morphological operations, binary template matching [9], thinning [10], convolution [7], etc. The straightforward parallel method for computing these iterative-convergent operators is through a globally synchronous updating mode: All variables are updated at once, based on the values calculated during the previous step, before another iteration step is initiated. Since systolic machines are designed to exploit spatial information and most of the spatial locality information is lost in compressed domain, most systolic image processing algorithms proposed so far are based on operations on pixel data. It is extremely difficult to design systolic algorithms which operate on compressed image data. Fortunately, some compression techniques such as RLE preserve part of the information pertaining to spatial locality allowing us to design a systolic system that finds the difference between two binary images represented in RLE.

In the next section, we elaborate on the RLE-based image difference algorithm. The following sections describe the parallel systolic system which computes the difference between the corresponding rows of two images represented in compressed form, i.e., RLE. (see Fig. 1). In Section 4, a formal proof of correctness for the systolic algorithm is provided. The last section gives simulation results for the systolic system which demonstrate that, for images with a high similarity measure, the time complexity of the systolic algorithm is small and in some cases constant with respect to the image size.

- *F. Ercal is with the Computer Science Department, University of Missouri, Rolla, MO 65401. E-mail: ercal@umr.edu.*
- *M. Allen is with Hewlett Packard, 3000 Waterview Parkway, Richardson, TX 75080. E-mail: markall@rsn.hp.com.*
- *H. Feng is with Microsoft, One Microsoft Way, Redmond, WA 98052. E-mail: haof@microsoft.com.*

Manuscript received 11 Sept.1998; accepted 9 Nov. 1999.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 107382.

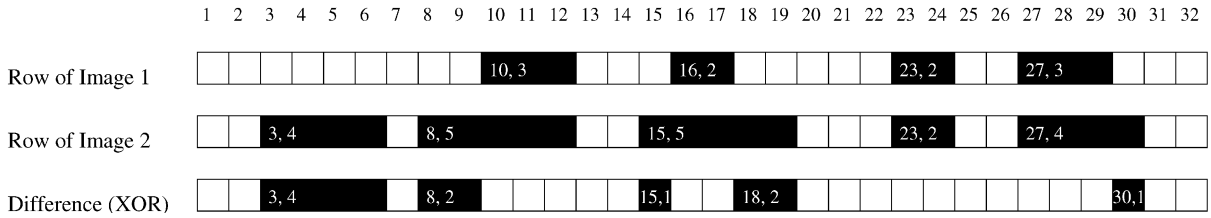


Fig. 1. Example of the image difference operation.

2 Image Difference

In this section, we provide a definition of the image difference problem and discuss a sequential algorithm to solve the problem on run-length encoded bitstrings.

Regardless of what encoding method is used, the inputs in the image difference problem both represent strings of binary data of the same length b. Let img1 and img2 be arrays representing these unencoded bitstrings of length b. Thus, for each location i in the range 1 to b, img1[i] has a value of one or zero based on whether image one has a foreground or a background-colored pixel in the ith location, respectively, and img2 is equivalently defined.

The output of the operation also represents a string of binary data of length b. The encoding of the output will matter later but not in the definition of the difference operation. Let difference be an array representing the unencoded output.

The desired output after an image difference operation is defined as follows:

Definition of Image Difference. *For each i in the range 1 to b, difference[i] = img1[i] $\oplus$ img2[i], where $\oplus$ represents the exclusive-or operation.*

An example image difference operation is shown in Fig. 1.

When using run-length encoding, the two inputs and the output are represented as arrays of 2-tuples of integers. In each tuple, the first element is the start of the run and the second element is the run's length. Each array of tuples must use a strictly increasing sequence of first elements of the tuples. By definition, none of the intervals represented by the tuples for a single bitstring may overlap. In the input it is permissible, in general, for two intervals in a single bitstring to be directly adjacent to each other, and in the output it is possible for this to occur as well; however, an additional pass can be made at the end to ensure the encoding is completely compressed. Note that only the foreground pixels are represented in the encoding.

The sequential algorithm for finding the image difference of two RLE encoded bitstrings is a single pass through the two arrays simultaneously, which merges them together into a single RLE encoded bitstring. We start at the beginning of the two arrays, and for each iteration, we determine the XOR of the top run of both bitstrings, take the smaller of the resulting runs, and leave the remainder in the array it came from. This sequential algorithm clearly has a time complexity of O(k) where k is the number of runs in the two images. Also, it should be noted that this time complexity is the same for the best, worst, and average case.

3 RLE-Based Systolic Image Difference Algorithm

If we let k be an upper bound on the number of runs in a single input bitstring, then the XOR operation can clearly not produce more than 2^*k runs, thus our systolic architecture will use 2^*k cells. Each cell will have two registers, each capable of storing two integers to represent a run, as shown in Fig. 2. Initially, the first register of each cell will be used to store the array of runs representing the first image, and the second register of each cell will store the array of runs for the second image. After the algorithm has terminated, the first register of the cells will represent the result of the XOR operation and the second register of all cells will be empty.

For notation, we will call the first register RegSmall and the second register RegBig. Also, we will refer to runs by their starting and ending points rather than the starting points and lengths which are actually stored. Thus, if cell i contains two runs, where the first one starts at location 10 and has length 5 and the second one starts at location 12 and has length 8, our notation will indicate this as

cell[i].RegBig.start = 10 cell[i].RegBig.end = 14
cell[i].RegSmall.start = 12 cell[i].RegSmall.end = 19

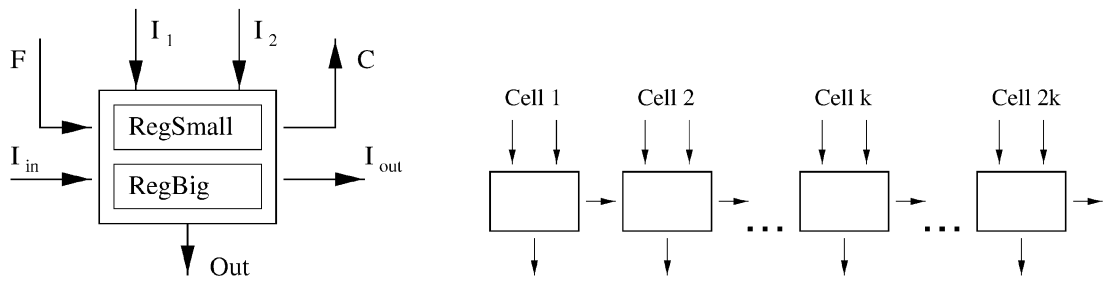


Fig. 2. Architecture of a cell, and array of cells forming the systolic system.

Iteration	Register	Cell1	Cell2	Cell3	Cell4	Cell5	Cell6
Initial	RegSmall	(10,3)	(16,2)	(23,2)	(27,3)		
	RegBig	(3,4)	(8,5)	(15,5)	(23,2)	(27,4)	
1 (step1)	RegSmall	(3,4)	(8,5)	(15,5)	(23,2)	(27,4)	
(step1)	RegBig	(10,3)	(16,2)	(23,2)	(27,3)		
1 (step2)	RegSmall	(3,4)	(8,5)	(15,5)	(23,2)	(27,4)	
(step2)	RegBig	(10,3)	(16,2)	(23,2)	(27,3)		
1 (step3)	RegSmall	(3,4)	(8,5)	(15,5)	(23,2)	(27,4)	
(step3)	RegBig	empty	(10,3)	(16,2)	(23,2)	(27,3)	
2 (step1)	RegSmall	(3,4)	(8,5)	(15,5)	(23,2)	(27,3)	
(step1)	RegBig	empty	(10,3)	(16,2)	(23,2)	(27,4)	
2 (step2)	RegSmall	(3,4)	(8,2)	(15,1)			
(step2)	RegBig	empty	empty	(18,2)	empty	(30,1)	
2 (step3)	RegSmall	(3,4)	(8,2)	(15,1)			
(step3)	RegBig	empty	empty	empty	(18,2)	empty	(30,1)
3 (step1)	RegSmall	(3,4)	(8,2)	(15,1)	(18,2)	empty	(30,1)
(step1)	RegBig	empty					
And steps 2 and 3 of iteration 3 make no further changes.							

Fig. 3. Execution of the systolic algorithm on the inputs from Fig. 1.

Now we will describe the main steps of the algorithm which will be put into a loop to form the final algorithm. These steps will be executed by each cell individually, and are written below to be executed by an arbitrary cell i.

Steps used in main algorithm.
1.) The purpose of this step is to put the "smaller" run into RegSmall and the "bigger" run into RegBig.

```
if (cell[i] has a run in both of its registers) then
  if ((cell[i].RegSmall.start > cell[i].RegBig.start) ||
    ((cell[i].RegSmall.start == cell[i].RegBig.start)  &&
       (cell[i].RegSmall.end > cell[i].RegBig.end))) then
      swap the contents of RegSmall and RegBig
    endif
else if (cell[i] has a run in only RegBig) then
    move the contents of RegBig to RegSmall and set
    RegBig to empty
endif
```

2.) Perform the XOR operation in cell i (independently from all other cells containing other runs). And to avoid any ambiguity as to where the resulting runs are stored in the cell, we can describe the XOR more explicitly. Each cell executes the following:

```
oldSmallend = RegSmall.end
RegSmall.end = min(RegSmall.end, RegBig.start-1)
RegBig.start = min(RegBig.end+1,
  max(oldSmallend+1, RegBig.start))
RegBig.end = max(oldSmallend, RegBig.end)
```

3.) Shift the data in RegBig to the right, and receive data from the left into RegBig.

Finally, we can put these three steps together into a loop to form the complete algorithm which is executed by each cell i.

Algorithm for cell i:

```
while (not receiving the termination signal along input F)
   do step-1 ; step-2 ; step-3 ;
   if (there is no data in RegBig) then
     send the termination signal along output C
   endif
endwhile
```

Externally, when all cells are sending the termination signal along output C, then the termination signal is sent along input F so that all the cells stop processing.

At this point, the runs stored along RegSmall in the cells form an array of runs which are ordered, do not overlap, and correctly represent the XOR of the original two bitstrings. A formal proof for this assertion is provided in the next section. Note that it is possible for there to exist empty cells between these runs, however. Fig. 3 illustrates the steps of a systolic run using the input from Fig. 1.

4 Proof of Correctness

There are three pieces to prove in this section. First, we must show that the algorithm does halt after a certain number of steps. Second, we must show that the resulting array of runs when the algorithm terminates is ordered and that none of the resulting sequences overlap. And third, we must show that the resulting array of runs does indeed represent the XOR of the original two bitstrings.

4.1 Proof for Termination

The first part is quite trivial to show by induction. We will use the following two corollaries which lead directly to our first theorem.

Corollary 1.1. *At the end of iteration i, the first i cells do not have any runs stored in RegBig.*

Corollary 1.2. *At no point in the algorithm will there exist a nonempty cell beyond location $k1 + k2$ where $k1$ is the number*

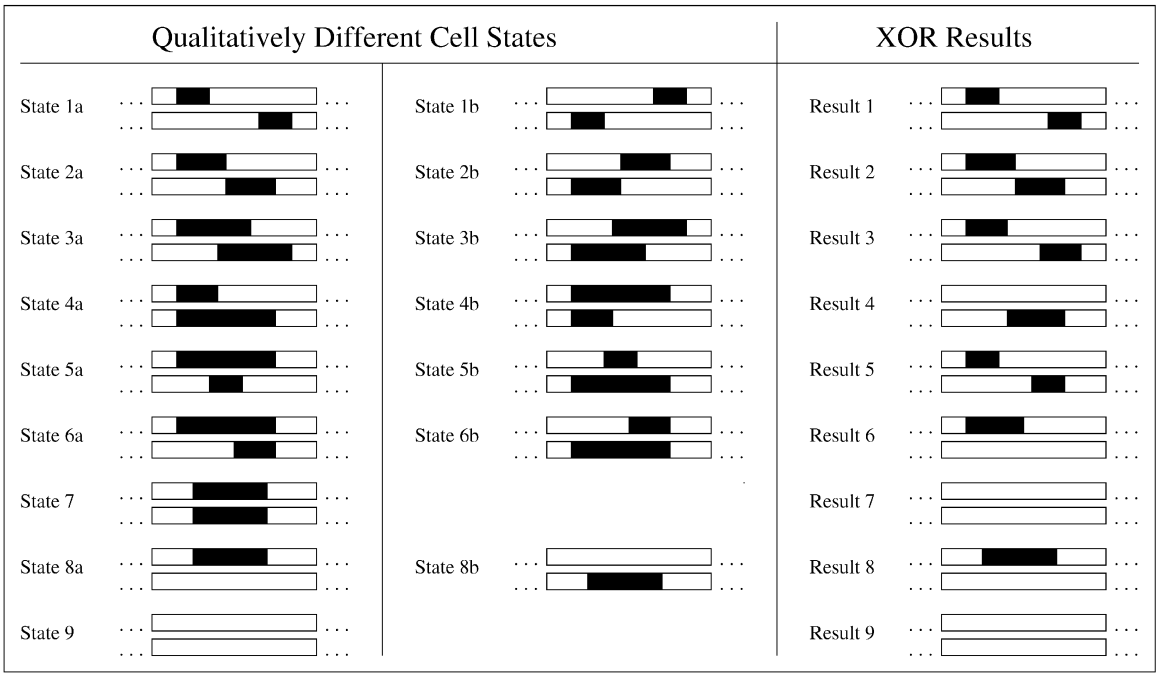


Fig. 4. List of qualitatively different cell states.

of runs in the first image and $k2$ is the number of runs in the second image.

Proof of Corollary 1.1. *Base case*: If we refer to the state before any iterations have been completed as the "end" of iteration 0, then the assertion is vacuously true.

To have a somewhat less trivial base case, we can also consider the state at the end of iteration 1. For this part, all we need to note is that Step 3 is the last step of the iteration. Thus, if there were anything in RegBig for cell 1 before Step 3, it would not be there after Step 3.

Induction step: Suppose that after iteration i it is the case that the first i cells do not have any runs stored in RegBig. We will now show that after the $i+1$st iteration, they still do not have any runs in RegBig and cell $i+1$ also has no run stored in RegBig.

Since none of the first i cells contain a run in RegBig, Step 1 where the smaller run is moved to the top will make no changes to the first i cells. Step 2 where the XOR is performed will also not cause any change. Step 3 where the data in RegBig is shifted right will not bring in any new data to these first i cells, nor will cell $i+1$ have any data in RegBig after Step 3. Therefore, the first $i+1$ cells have no data in RegBig after the $i+1$st iteration. □

Proof of Corollary 1.2. Note first that the total number of runs in the system never increases. Step 2 may sometimes reduce two runs into one, but none of the steps will ever produce more runs than the step started with. Next, note that only Step 3 is capable of causing the location of the rightmost nonempty cell to increase, and when this happens it results in a run being moved from RegBig to RegSmall in Step 1 at the beginning of the next iteration, thus decreasing the number of runs stored in RegBig across all the cells. And since no steps cause the number of runs stored in RegBig of the cells to increase, this increase in the location of the rightmost nonempty cell can only happen a limited number of times.

More specifically, at the start of the algorithm there are $k1$ cells containing a run in RegSmall and $k2$ cells containing a run in RegBig. After Step 1 of the first iteration, there are $\max(k1, k2)$ cells containing a run in RegSmall and $\min(k1, k2)$ cells containing one in RegBig. At this point, the rightmost nonempty cell is at location $\max(k1, k2)$, and based on the previous discussion, this can only increase $\min(k1, k2)$ times to a maximum value of $\max(k1, k2) + \min(k1, k2)$, which clearly reduces to $k1 + k2$. □

Now that the two corollaries are proven, we can restate our theorem about termination.

Theorem 1. *The systolic XOR algorithm terminates after at most $k1 + k2$ steps, where k_1 is the number of runs in the first image and k_2 is the number of runs in the second image.*

Proof of termination. By Corollary 1.1, after iteration $k1 + k2$, the first $k1 + k2$ cells have no runs stored in RegBig. By Corollary 1.2 there are no nonempty cells beyond location $k1 + k2$. Thus, by iteration $k1 + k2$ the only nonempty cells are ones which have no runs stored in RegBig, which means that the termination condition is satisfied by iteration $k1 + k2$. □

4.2 Proof for Proper Ordering

In this section, we prove that the resulting array of runs when the algorithm terminates is ordered and that none of the resulting sequences overlap. This part takes somewhat longer to prove than the termination. First, we will introduce some more notation to be able to refer qualitatively to all the various possible states a cell can be in. These states are shown in Fig. 4.

The first two columns of Fig. 4 show all the possible cell states, and the third column shows the result of performing Steps 1 and 2 on each of these cells. The reason for the pairings between columns 1 and 2 is that the "a" states and the "b" states are related in the sense that any "b" state will turn into the corresponding "a" state after Step 1 is performed, and any "a" state will be unchanged by a Step 1.

We wish to prove that the runs stored along RegSmall and RegBig of the cells are always ordered. More specifically, we show the following.

Theorem 2. *At the end of each iteration, for every cell i, and every cell j to its right $(j > i)$,*

1. *if both cells i and j contain runs in RegSmall, then cell[i].RegSmall.end $<$ cell[j].RegSmall.start, and*
2. *if both cells i and j contain runs in RegBig, then cell[i].RegBig.end $<$ cell[j].RegBig.start.*

We can write the theorem in a format more conducive to proof as follows. Since each iteration of the algorithm consists of three steps and the third is so simple, we focus the corollary below on the first two steps. For notation, we refer to the state of cell i before an iteration begins as cell[i].before, and the state of the cell after the first, second, and third steps as cell[i].after1, cell[i].after2, and cell[i].after3, respectively. Note that the current iteration is not included because it would unduly clutter the notation. Thus, the iteration being considered must be made clear from context.

Corollary 2.1. *At any iteration, for every cell i, and for every cell j to its right $(j > i)$,*

1. *if both cells i and j contain runs in RegSmall after Step 2, then*

$$cell[i].after2.RegSmall.end < cell[j].after2.RegSmall.start,$$

2. *if both cells i and j contain runs in RegBig after Step 2, then*

$$cell[i].after2.RegBig.end < cell[j].after2.RegBig.start,$$

3. *if cell i has a run in RegSmall and in RegBig after Step 2, then*

$$cell[i].after2.RegSmall.end < cell[i].after2.RegBig.start,$$

and

4. *if cell i has a run in RegSmall and cell j has one in RegBig after Step 2, then*

$$cell[i].after2.RegSmall.end < cell[j].after2.RegBig.start.$$

5. *If after Step 3 some cell k between cells i and j (including i itself) has no run in RegSmall, and if cell i has a run in RegBig and cell j has a run in RegSmall, then*

$$cell[i].after3.RegBig.end < cell[j].after3.RegSmall.start.$$

Note that parts three, four, and five of the above corollary are included only because they are useful in proving the induction step. The proof of Corollary 2.1 is by induction on the number of iterations and it is provided in the Appendix. The first four parts are reasonably intuitive; however, the fifth part may not be. In the proof given in the Appendix, the first four parts follow rather directly from some simple inequalities, while the fifth part requires more reasoning.

Once Corollary 2.1 is proven, it is fairly easy to show Theorem 2:

Proof of Theorem 2. Execution of Step 3 of the algorithm does not have any effect on the truth of the first part of Corollary 2.1. Thus, if the first inequality from Corollary 2.1 is shown to be true between cells i and j after Steps 1 and 2 are performed, then the first part of Theorem 2 is true, too. If part two of the corollary is shown to be true between cells i and j after Steps 1 and 2, then part two of the theorem is true for all cells $i+1$ and $j+1$, which covers all pairings which do not use the first cell. And since RegBig of this first cell is empty, the pairings involving it are vacuously true. □

4.3 Correctness Proof for the Resulting RLE String

To conclude the formal proof of correctness for our systolic algorithm, we need to show that the resulting array of runs does indeed represent the XOR of the original two bitstrings. This part is rather easy compared to the previous section. The idea is to view the runs of the two bitstrings as a set of many distinct smaller bitstrings and observe that the only changes made to this set involve XORs among these bitstrings. This, combined with the fact that XOR is associative, implies that the final state is the correct XOR of the original two bitstrings.

In more detail, the definition of the image difference problem was given as difference[i] = img1[i] $\oplus$ img2[i], for each i in the range 1 to b, where $\oplus$ represents the exclusive-or operation, and where b is the number of pixels in the image.

We can easily extend this to apply to a set of bitstrings instead of merely two bitstrings. We could write this as

$$difference[i] = \begin{cases} 0 & if\ an\ even\ number\ of\ bitstrings\ from\ our\ set \\ & \quad have\ a\ one\ in\ bit\ i,\ or \\ 1 & if\ an\ odd\ number\ of\ bitstrings\ from\ our\ set \\ & \quad have\ a\ one\ in\ bit\ i. \end{cases}$$

For two bitstrings, these are clearly equivalent definitions of the difference. For any set of bitstrings, we will view the difference of the entire set according to the definition above.

To make this definition useful we must make the observations that

Corollary 3.1. *if the runs of a bitstring are viewed as a set of smaller bitstrings, then the XOR of this set is the original bitstring, and*

Corollary 3.2. *letting xor(A) represent the bitstring which results from XORing the bitstrings contained in the set A, we have for arbitrary sets of bitstrings A and B that xor(A ∪ B) = xor({xor(A), xor(B)}).*

Proof of Corollary 3.1. The first observation follows from the fact that none of the runs in the original bitstring can overlap, therefore each location in the difference array will be a one if the original bitstring had a one, and a zero if the original had a zero in the specified location. □

Proof of Corollary 3.2. The second observation is true because for each location in the difference array,

- if a resulting bit in xor(A ∪ B) is a one, then the total number of runs between A and B that had a one in that location must have been odd. Therefore, one of the sets had an odd number of ones and the other must have had an even number, and between xor(A) and xor(B) one has a one and the other has a zero. Thus, xor({xor(A), xor(B)}) has a one in the specified bit,
- and if a resulting bit in xor(A ∪ B) is a zero, then the total number of runs between A and B that had a one in that location must have been even. Therefore, either both sets had an odd number of ones, or they both had an even number of ones in the specified bit. Thus, xor(A) and xor(B) either both have a one or both have a zero, and either way this causes xor({xor(A), xor(B)}) to have a zero in the specified bit. □

Now we wish to use these corollaries to prove that the image difference produced by the algorithm is correct.

Theorem 3. *The image difference produced by the systolic algorithm is the same as the correct XOR defined in Section 2.*

Proof of Correctness. We can let A be the set of runs contained in the first image, and let B be the set of runs in the second image. Thus, based on our first observation, xor(A) is the first image and xor(B) is the second image, so the final result we seek is xor({ xor(A), xor(B)}), which according to our second observation is equal to xor(A ∪ B).

Now that we have expressed the desired result as an XOR over the set of all runs contained in the two images, we must show that although the set of runs being considered changes at each step of the algorithm, the resulting XOR is still the same after each iteration.

Clearly, Steps 1 and 3 of a given iteration do not change the set of runs under consideration. Only the second step causes any changes. And since XOR is an associative operation, we can say that xor(A ∪ B) is xor(A ∪ {xor(B)}) by an argument very similar to the one used in our second observation above. Letting B be a pair of runs XORed in a cell during Step 2, we see that the XOR of the set of runs before Step 2 is the same as the XOR of the new set of runs after Step 2. Thus, we have now shown that at any point in the algorithm, if C is the set of runs contained in the systolic system, then xor(C) is the correct XOR (i.e., xor({xor(A), xor(B)})). And due to Theorems 1 and 2, when the iterations are over, the final result will be stored in RegSmall in a sorted and nonoverlapping manner, thus making xor(C) equal to the bitstring represented directly by the runs of C. That is, the bitstring stored in the end is indeed the correct XOR. □

5 Algorithm Performance

In this section, we present experimental results to show that the systolic algorithm obtains the final result very quickly when the bitstrings being XORed are highly similar.

First, another upper bound can be put on the number of steps the algorithm will take. When we proved termination above, we showed it would stop in at most $k1 + k2$ steps where $k1$ is the number of runs in the first bitstring, and $k2$ is the number of runs in the second bitstring. As stated below, we also believe that it is bounded by the number of runs in the image difference, although we have not yet proven this.

Observation. If the runs of the two input bitstrings are encoded such that none of the runs are adjacent (in other words, if the bitstring is compressed as much as possible), then the systolic XOR algorithm terminates after at most $k3 + 1$ steps, where $k3$ is the number of runs in the output from the systolic algorithm (note the output from the systolic algorithm will not always be compressed optimally).

If we let the similarity of two images be measured by the number of runs in the final result, then the above observation implies that the systolic algorithm has the potential to run faster the more similar two bitstrings are.

A simulation program was written to test the algorithm on a large number of randomly generated input cases. The size for the image rows was varied from 128 to 2048 pixels. The "on" pixels in the first image were chosen in runs of length 4 to 20, and the second image was obtained by flippping some of the bits of the first image in either direction ($1 \rightarrow 0$, $0 \rightarrow 1$). Here, these changes are called "errors" and they were in runs of length 2 to 6. The percentage of "on" pixels in the first image and of the errors in the second image was varied by changing the average distance between the runs.

The empirical testing shows that for medium amounts of error (when the number of pixels changed was less than 30 percent of the total image) the dominating factor for the number of iterations was the difference between the number of runs in the two images. This was true irrespective of the sizes of the images and varied only slightly over different densities.

This is demonstrated in Fig. 5, which shows the average number of iterations taken by the algorithm as a function of the percentage of pixels with errors. In this figure, the image size is 10,000 pixels with approximately 250 runs in the original image, which translates to a density of 30 percent. The pattern is similar for smaller images, but the variation is higher. The other two sets of data show the average difference in the number of runs in the two images, which

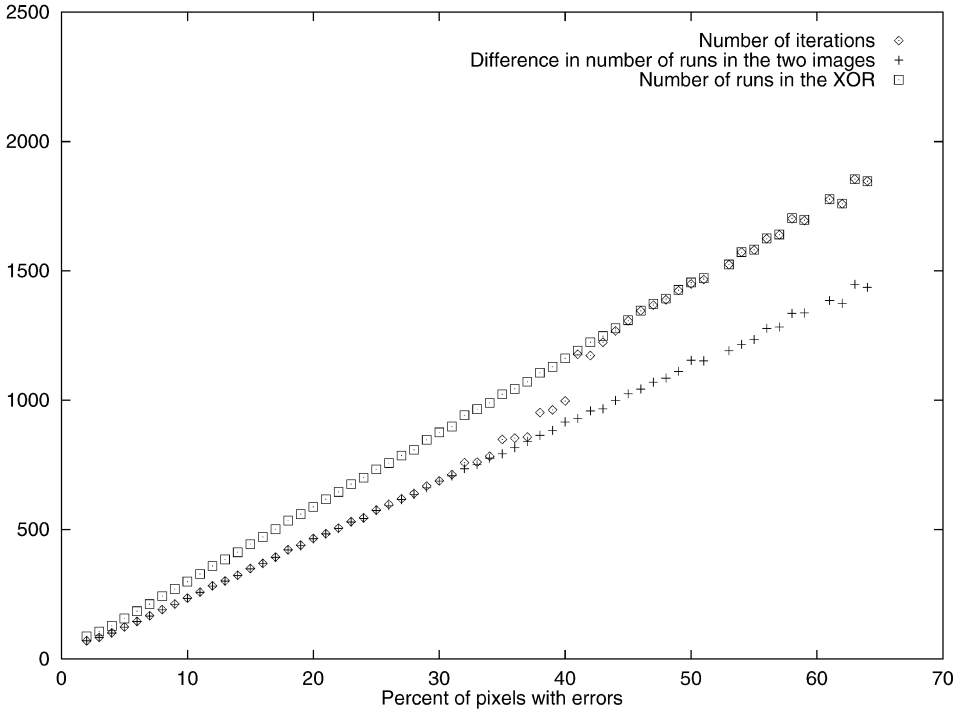


Fig. 5. Number of iterations as a function of the percent of pixels with errors plotted along side two of the dominating factors in the algorithm's running time.

correlate very closely with the number of iterations up through 30-40 percent, and the number of runs in the XOR produced by the algorithm which is the upper bound we have not proven yet.

In explanation of the high correlation between the number of iterations taken and the difference in the number of runs in the two images, we notice that after the first iteration the larger number of runs will be stored along RegSmall. Then if the shift-right procedure in Step 3 causes a run to be pushed into this group of runs along the end, then all the runs at the end will need to be pushed to the right a cell. And clearly the number of steps taken by this chain reaction will be the length of this group of runs at the end, which is the difference between the number of runs in the two images.

When the number of pixels changed is much greater than 30 percent of the total image, a different factor begins to dominate. For the smaller amounts of difference, there will be lots of empty cells left behind throughout the array, thus the only significant data movement will be at the end as discussed in the previous paragraph. But as the number of differences increases and thus, the number of empty cells decreases, more and more data movement will be required thus pushing the algorithm closer to the upper bound.

Fig. 5 demonstrates the correlation between the number of iterations taken by the algorithm and the difference in the number of runs in the two images and it demonstrates an upper bound as the number of runs in the XOR after the algorithm finishes; however, it does not give a good impression of the algorithm's speed. This can be seen in the next table which focuses on smaller amounts of error.

Table 1 shows the average number of iterations taken by both the sequential and the systolic algorithm on an image of size ranging from 128 to 2,048 pixels. In the first case, the errors are kept at approximately 3.5 percent of the image, thus causing both the systolic and the sequential versions to take linearly more time as the image size increases. In the second case, however, the number of errors is fixed at six runs each of size 4 pixels, thus while the sequential algorithm still takes large amounts of time, the systolic algorithm averages just over six iterations regardless of how large the image gets.

6 Conclusions and Future Research

This paper has shown that a systolic array can perform an image difference operation on RLE encoded images very quickly if the two images are highly similar. Indeed, the number of iterations taken is bounded above by the number of runs left in the XOR, and for similar images, the number of iterations is tightly correlated with the difference between the number of runs in the two images.

TABLE 1
Average Systolic Iterations versus Sequential Iterations for Small Amounts of Errors (Length of Runs: 4-20, Length of Error Runs: 2-6)

Algorithm	Errors	Iterations versus image size				
		128	256	512	1024	2048
Systolic	3.5%	2.0	3.1	5.3	9.7	17.3
Sequential	3.5%	8.9	17.5	34.5	68.9	137.3
Systolic	6 runs	6.6	6.2	6.3	6.4	6.4
Sequential	6 runs	13.4	20.6	35.6	65.8	126.0

Although a parallel solution of the image difference problem can easily be performed on uncompressed data in constant time if the number of processors available is proportional to the number of pixels in the images, there is no known parallel algorithm which performs the same operation in compressed mode. To the best of our knowledge, this paper demonstrates the first effective parallel solution which operates on compressed data directly. This method has the advantage of using a smaller number of processors, and it does not require the time to convert between RLE format and the bitmap mode when the original image is in RLE format. Here, we note that, in any practical application, k will be a lot larger than the number of systolic cells available. To solve this problem of disparity between the number of runs and the PEs, for each computation phase, at most k runs are loaded to the systolic cells. To do this effectively, the image rows must be partitioned into sections of equal length (in pixels) in such a way that in each section one of the image rows will contain k runs while the other image row will contain less than k runs covering the same number of image pixels. This partitioning phase can be overlapped with the systolic computation, or marking of the image rows can be done beforehand during the image acquisition phase. Another important issue to mention here is how to perform input/output between the systolic cells and the image memory in a cost effective manner. Our timing simulations suggest that this problem can be solved by overlapping I/O with the XOR computation. Indeed, this overlapping strategy is adopted in our current implementation of the systolic array [13] which uses Field Programmable Gate Arrays (FPGAs) [14].

In both the case of highly similar and highly different images, the number of iterations taken seems to be dominated by the frequent need to push a whole set of runs to the right to make room for a new entry. If a broadcast bus existed which could run at the same frequency as the rest of the systolic system, it might be possible to perform these shifts more efficiently, thus significantly decreasing the running time. Thus, one area of future research should be modifying the algorithm to run more quickly on a model with a fast broadcast bus, such as a reconfigurable mesh [12]. Additionally, the task of combining the adjacent runs in different cells at the end of the algorithm is left as a future research. This task also is not fast on a pure systolic system, but could be performed quickly with the help of a broadcast bus. We are also working on a proof to show that our observation stated in Section 5 does indeed hold.

APPENDIX

In the induction proof below, we will consider cell i and cell j in any of the possible states shown in Fig. 4 and show that the five parts of Corollary 2.1 hold for each possible pairing. Fortunately, these possibilities can be grouped together into a relatively small number of cases. For notation, we refer to the state of cell i before an iteration begins as cell[i].before, and the state of the cell after the first, second, and third steps as cell[i].after1, cell[i].after2, and cell[i].after3, respectively. Note that the current iteration is not included because it would unduly clutter the notation. Thus, the iteration being considered must be made clear from context.

In the inequalities proven below for each pairing, certain observations will be used several times. The following inequalities can be verified by examining Fig. 4.

Cells 1a-3a, 1b-3b, 5a, 5b, 6a, 6b, and 8a:

$$cell.after2.RegSmall.end \leq cell.before.RegSmall.end$$

Cells 1a-3a, 5a, 6a, and 8a:

$$cell.before.RegSmall.start = cell.after2.RegSmall.start$$

Cells 1a-3a, 1b-3b, 5a, 5b, 6a, 6b, and 8b:

$$cell.after2.RegSmall.end \leq cell.before.RegBig.start$$

Cells 1b-3b, 5b, 6b, and 8b:

$$cell.before.RegBig.start = cell.after2.RegSmall.start$$

Cells 1a-4a, and 5b:

$$cell.after2.RegBig.end = cell.before.RegBig.end$$

Cells 1a-5a, and 1b-5b:

$$cell.before.RegBig.start \leq cell.after2.RegBig.start$$
$$cell.before.RegSmall.start \leq cell.after2.RegBig.start$$

Cells 1b-4b, and 5a:

$$cell.after2.RegBig.end = cell.before.RegSmall.end.$$

Now, we are finally ready for the induction proof on the number of iterations.

Proof of Corollary 2.1

Base case. We must show that after Steps 1 and 2 of iteration 1, for all cells i and j where $j > i$, the first four inequalities of Corollary 2.1 hold for all possible pairings of states that cells i and j could have. And, we must show that the fifth part is true for any possible pairing after the third step, too.

Note that since the initial configuration has no holes, the following pairings are not possible:

- state 8a followed by any other than 8a or 9
- state 8b followed by any other than 8b or 9
- state 9 followed by any other than 9.

Now, we show the inequalities for all other pairings. Note that the actual assertion being shown in all cases is

> if cell[i].after.AppropriateRegister and
> cell[j].after.AppropriateRegister both contain runs, then
> the stated inequality holds.

And the inequality is shown using the list that was developed immediately before the proof, plus the obvious inequalities that follow from the fact that the runs in the before state (which is the initial input to the problem) are ordered. That is,

$$cell[i].before.RegSmall.end < cell[j].before.RegSmall.start$$
$$cell[i].before.RegBig.end < cell[j].before.RegBig.start.$$

- Inequality 1:
  - any state but 8b followed by 1a-6a, 7, 8a, or 9:

$$\begin{aligned} &cell[i].after2.RegSmall.end \\ &\leq cell[i].before.RegSmall.end \\ &< cell[j].before.RegSmall.start \\ &= cell[j].after2.RegSmall.start \end{aligned}$$

  - any state but 8a followed by 1b-6b, or 8b:

$$\begin{aligned} &cell[i].after2.RegSmall.end \\ &\leq cell[i].before.RegBig.end \\ &< cell[j].before.RegBig.start \\ &= cell[j].after2.RegSmall.start \end{aligned}$$

  - state 8a followed by 8a or 9: trivial
  - state 8b followed by 8b or 9: trivial
  - Inequality 2:
    - state 1a-4a, 5b, 6a, 6b, 7, 8a, 8b, or 9 followed by any:

$$\begin{aligned} &cell[i].after2.RegBig.end \\ &= cell[i].before.RegBig.end \\ &< cell[j].before.RegBig.start \\ &\leq cell[j].after2.RegBig.start. \end{aligned}$$

    - state 1b-4b, or 5a followed by any:

$$\begin{aligned} &cell[i].after2.RegBig.end \\ &= cell[i].before.RegSmall.end \\ &< cell[j].before.RegSmall.start \\ &\leq cell[j].after2.RegBig.start. \end{aligned}$$

  - Inequality 3: This is much simpler than the previous steps. A simple inspection of all possible states in the third column of Fig. 4 shows that in all cases

$$\begin{aligned} &cell[i].after2.RegSmall.end \\ &< cell[i].after2.RegBig.start. \end{aligned}$$

  - Inequality 4: This doesn't quite follow directly from parts 2 and 3 because cell i might not have a run in RegBig. The only times it will have a run in RegSmall, but not RegBig is if it starts in state 6a, 6b, 8a, or 8b. Since both registers in states 6a and 6b have the same ending point, we easily get that for any cell j to the right,

$$\begin{aligned} &cell[i].after2.RegSmall.end \\ &\leq cell[i].before.RegSmall.end \\ &= cell[i].before.RegBig.end \\ &< cell[j].before.RegBig.start \\ &\leq cell[j].after2.RegBig.start. \end{aligned}$$

And for states 8a and 8b, we have the advantage that 8a cannot be followed by anything other than 8a or 9, and 8b cannot be followed by anything other than 8b or 9. Thus, the "after" state of cell j will never have a run in RegBig to worry about.

- Inequality 5: For some cell k between i and j (including i) to end up with no run in RegSmall after the iteration is complete, it must have started in state 4a, 4b, 7, or 9. Since state 9 cannot be followed by anything but another cell in state 9 at the start of the first iteration, our inequality is vacuously true for that state.

  Focusing on cell k having state 4a, 4b, or 7 initially, we see that the starting points of the two runs in cell k form a nice barrier for any runs in any cells preceding k. And naturally, the run which ends up in cell[i].RegBig after the iteration comes from behind this barrier (or if it doesn't, then our inequality is vacuously true again).

  More specifically, we have

$$\begin{aligned} &cell[i].after3.RegBig.end \\ &= cell[i-1].after2.RegBig.end, \end{aligned}$$

which is in turn equal to either cell[i-1].before.RegSmall.end or cell[i-1].before.RegBig.end. And either way, we have

$$\begin{aligned} &cell[i-1].before.RegSmall.end \\ &\quad < cell[k].before.RegSmall.start \\ &cell[i-1].before.RegBig.end \\ &\quad < cell[k].before.RegBig.start, \end{aligned}$$

which gives us

$$\begin{aligned} &cell[i].after3.RegBig.end \\ &= cell[i-1].after2.RegBig.end \\ &< cell[k].before.RegSmall.start \\ &= cell[k].before.RegBig.start. \end{aligned}$$

And since cell j is to the right of k, we know that both

$$\begin{aligned} &cell[k].before.RegSmall.end \\ &< cell[j].before.RegSmall.start \end{aligned}$$

and

$$\begin{aligned} &cell[k].before.RegBig.end \\ &< cell[j].before.RegBig.start \end{aligned}$$

and one of these is in turn

$\leq$ cell[j].after2.RegSmall.start,

which, being in RegSmall, is the same as cell[j].after3.RegSmall.start.

Induction step: Now suppose we know that during the steps of the kth iteration it is the case that for any cells i and j where $j > i$, the five inequalities of Corollary 2.1 hold for all possible pairings of states that cells i and j could have. We must show that the first four inequalities all still hold after the first two steps of the $k+1$st iteration, and the fifth one holds after the third step.

Note that we cannot rule out any pairings this time, so we must use the extra information provided in the induction step to prove the inequalities for all pairings.

Also, note that the actual assertion being shown in all cases is

if cell[i].after.AppropriateRegister and
cell[j].after.AppropriateRegister both contain runs, then
the stated inequality holds.

And the inequality is shown using the list that was developed immediately before the proof, plus the inequalities that follow from the induction hypothesis.

- Inequality 1:
  - any state but 8b followed by 1a-6a, 7, 8a, or 9: same as base case
  - any state but 8a followed by 1b-6b, or 8b: same as base case
  - 8a followed by 1b-6b, or 8b:

$$\begin{aligned} &cell[i].after2.RegSmall.end \\ &= cell[i].before.RegSmall.end \\ &< cell[j].before.RegBig.start \\ &= cell[j].after2.RegSmall.start. \end{aligned}$$

  - 8b followed by 1a-6a, 7, 8a, or 9:

$$\begin{aligned} &cell[i].after2.RegSmall.end \\ &= cell[i].before.RegBig.end \\ &< cell[j].before.RegSmall.start, \end{aligned}$$

since there exists k between i and j (specifically $k = i$) such that cell k has no run in RegSmall at the end of the previous iteration. And continuing, we have

$$\begin{aligned} &cell[j].before.RegSmall.start \\ &= cell[j].after2.RegSmall.start. \end{aligned}$$

  - Inequality 2: The exact same reasoning used in the base case applies here too. Conveniently, no possible pairings were skipped.
  - Inequality 3: Again, the same reasoning used in the base case applies.
  - Inequality 4: Here, we can again use the information contained in the induction hypothesis to our advantage. Based on part 1, we know that since cell j is to the right of i,

$$\begin{aligned} &cell[i].before.RegSmall.end \\ &< cell[j].before.RegSmall.start. \end{aligned}$$

And putting this together with the other known inequalities we have that

  - any state but 8b followed by any:

$$\begin{aligned} &cell[i].after2.RegSmall.end \\ &\leq cell[i].before.RegSmall.end \\ &< cell[j].before.RegSmall.start \\ &\leq cell[j].after2.RegBig.start. \end{aligned}$$

  - state 8b followed by any:

$$\begin{aligned} &cell[i].after2.RegSmall.end \\ &= cell[i].before.RegBig.end \\ &< cell[j].before.RegBig.start \\ &\leq cell[j].after2.RegBig.start. \end{aligned}$$

  - Inequality 5: For some cell k between i and j (including i) to end up with no run in RegSmall after the iteration is complete, it must have started in state 4a, 4b, 7, or 9. Our argument from the base case still works for states 4a, 4b, and 7, so we now need only argue that the inequality still holds if cell i starts in state 9 at the beginning of this iteration.

    In the same way that the starting points of the two runs in cell k formed a nice barrier in the other states, it forms the same barrier in state 9 here due to the induction hypothesis.

    More specifically, we have

$$\begin{aligned} &cell[i].after3.RegBig.end \\ &= cell[i-1].after2.RegBig.end \end{aligned}$$

which is in turn equal to either cell[i-1].before.RegSmall.end or cell[i-1].before.RegBig.end. And either way, we have

$$\begin{aligned} &cell[i-1].before.RegSmall.end \\ &< cell[j].before.RegSmall.start, and \\ &cell[i-1].before.RegSmall.end \\ &< cell[j].before.RegBig.start \end{aligned}$$

due to part 1 and parts 3 and 4 of the induction hypothesis, and

$$\begin{aligned} &cell[i-1].before.RegBig.end \\ &< cell[j].before.RegSmall.start \\ &cell[i-1].before.RegBig.end \\ &< cell[j].before.RegBig.start \end{aligned}$$

due to part 2 and part 5 of the induction hypothesis. And this gives us

$$
\begin{aligned}
&cell[i].after3.RegBig.end \\
&= cell[i-1].after2.RegBig.end \\
&< cell[j].before.RegSmall.start, and \\
&cell[i].after3.RegBig.end \\
&= cell[i-1].after2.RegBig.end \\
&< cell[j].before.RegBig.start
\end{aligned}
$$

and one of these is in turn

$$\leq \text{cell[j].after2.RegSmall.start},$$

which, being in RegSmall, is the same as cell[j].after3.RegSmall.start. □

ACKNOWLEDGMENTS

This work was supported in part by the Department of Navy (Contract # N00164-97-C-0008) through a subcontract from Automation Engineering Inc. The authors would like to thank Robert Bible Jr., Jodi Bible, and Dr. Mark Burnett from AEI as well as anonymous referees for many helpful comments.

REFERENCES

[1] P.P. Jonker and E.R. Komen, "A Scalable Real-Time Image Processing Pipeline," *Proc. 11th Int'l Conf. Pattern Recognition (IAPR),* vol. 4, 1992. *Conference D: Architectures for Vision and Pattern Recognition,* pp. 142–146.

[2] F. Ercal, F. Bunyak, F. Hao, and L. Zheng, "A Fast Modular RLE-Based Inspection Scheme for PCBs," *Proc. Int'l Society for Optical Eng., (SPIE)—Architectures, Networks, and Intelligent Systems for Manufacturing Integration,* vol. 3,203, pp. 49–59, Oct. 1997.

[3] S. Gil, R. Milanese, and T. Pun, "Comparing Features for Target Tracking in Traffic Scenes," *Pattern Recognition,* vol. 29, no. 8, pp. 1,285–1,296, 1996.

[4] H. Kawasumi, H. Sekii, N. Enomoto, H. Ohata, and A. Okazaki, "Detecting Intruders using Time-Series Data by Projection Pattern of Silhouette," *Electrical Engineering in Japan,* vol. 119, no. 1, pp. 62–73, 1997.

[5] G.M. Emelyanov, N.V. Kurmyshev, and O.Y. Yuvzhik, "Procedures and Algorithms for Detecting and Determining the Orientation of Objects in Binary Images," *Pattern Recognition and Image Analysis,* vol. 7, no. 3 pp. 373–378, 1997.

[6] G. Agam, J. Frydman, O. Amiram, and I. Dinstein, "Efficient Morphological Processing of Maps and Line-Drawings Based on Directional Interval Coding," *Proc. SPIE—The Int'l Soc. for Optical Eng.,* vol. 3,168 pp. 41–51, 1997.

[7] N.K. Ratha, A.K. Jain, and D.T. Rover, "Convolution on Splash 2," *Proc. IEE Symp. FPGAs for Custom Computing Machines,* Apr. 1995.

[8] A. Rasquinha and N. Ranganathan, "C3L: A Chip for Connected Component Labelling," *IEEE 10th International Conf. VLSI Design,* pp. 446–51, Jan. 1997.

[9] M. Djunatan and T. Mengko, "A Programmable Real-Time Systolic Processor Architecture for Image Morphological Operations, Binary Template Matching and Min/Max Filtering," *1991 IEEE Int'l Symp. Circuits and Systems,* vol. 1, pp. 65–68, 1991.

[10] N. Ranganathan and K.B. Doreswamy, "A Systolic Algorithm and Architecture for Image Thinning," *Proc. 5th Great Lakes Symp. VLSI,* Mar 1995.

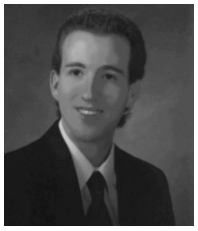
[11] Vipin Kumar, A. Grama, A. Gupta, and G. Karypis, *Intro. to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings Publishing Company Inc., 1994.

[12] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster, "The Power of Reconfiguration," *J. Parallel Distributed Computing,* vol. 13, pp. 139–153, 1991.

[13] F. Ercal, H. Pottinger, V.S. Balakrishnan, and M. Agarwal, "Design and Implementation of a Systolic Array Based RLE Image Processor Using an FPGA," http://vision1.cs.umr.edu/~systolic/Project/detail.html.

[14] J.V. Oldfield and R.C. Dorf, *Field Programmable Gate Arrays.* John Wiley and Sons, Inc., 1995.

Fikret Ercal received the BS (with first-class honors) and MS degrees in electrical engineering from Istanbul Technical University, and the PhD degree in computer science from the Ohio State University in 1988. Between 1988-1990, he was an assistant and associate professor at Bilkent University, Ankara. He is currently a professor at UMR and the director of the computer vision laboratory. Dr. Ercal is the recipient of three Faculty Excellence Awards from UMR. He is an associate editor of the *International Journal of Parallel and Distributed Systems and Networks*, executive committee member and the newsletter editor for the *IEEE Technical Committee on Parallel Processing*. He has published extensively in the areas of parallel computing, computer vision, and image processing.



Mark Allen received the BS degree in mathematics and the MS in computer science from the University of Missouri at Rolla in 1996 and 1998, respectively. He currently works for Hewlett Packard in Richardson, Texas. His research interests include parallel processing, reconfigurable computing, and quantum computing.

Hao Feng received the BS from Northwestern Polytechnical University, Xi'an, China, and MS from University of Missouri-Rolla in computer science in 1995 and 1998, respectively. Currently, he works for Microsoft in Redmond, Washington. His research interests include parallel algorithms, reconfigurable computing, and image proccessing.