Cache Tiling for High Performance Morphological Image Processing

Craig M. Wittenbrink Arun K. Somani†

Dept. of Electrical Engineering,
(and Dept. of Computer Science and Engineering)†
University of Washington
MS FT-10, Seattle, WA USA 98195

Appears as: Cache tiling for high performance morphological image processing, Craig M. Wittenbrink and A. K. Somani. In Journal *Machine Vision and Applications*, Vol. 7 No. 1, Winter 1993, pages 12-22. Short version in CAMP 91, Computer Architecture For Machine Perception, Paris, France, 1991, pages 427-438.

Author's Biographies:

Craig M. Wittenbrink received his B.S. in electrical engineering and computer science from the University of Colorado, Boulder CO in 1987, and a M.S. in electrical engineering from the University of Washington, Seattle WA in 1990. He is a Ph.D. candidate at the University of Washington studying parallel volume rendering with support from a NASA fellowship. He was a digital design engineer with Boeing Aerospace from 1987 to 1989 working on firmware design and testing for computer image generation hardware. Mr. Wittenbrink's primary research interests include computer graphics, computer architecture, and image processing.

Arun K. Somani received the B.E. (Hons.) degree in electronics engineering from Birla Institute of Technology and Science, Pilani, India in 1973, and the M.Eng. and Ph.D. degrees in electrical engineering from McGill University, Montreal, Canada, in 1983 and 1985, respectively. From 1974 to 1982, he worked as a Scientist in the System Group, Department of Electronics, Government of India, New Delhi. Currently he is an Associate Professor of Electrical Engineering and Computer Science and Engineering at the University of Washington, Seattle, WA. Dr. Somani's research interests include computer architecture, fault tolerant computing, parallel computer systems and algorithms, and computer interconnection networks. He is the architect of Proteus Computer System, a large reconfigurable network of processors for image and vision processing applications being built at the University of Washington.

Abstract

Morphological image analysis is a technique of processing images through shape characteristics (Jain 1989). Because images are regular data structures morphology algorithm's memory access patterns are predictable. By using read and write patterns, we derive a model of processing to examine inefficiencies for cache processing. We then develop a cache architecture for windowed processing that reduces cache thrashing. Our caching technique, cache tiling, allows dramatic improvement in caching efficiency for small caches independent of compiler optimizations. Programs are not affected providing a transparent solution to improve caching. System code, compilers, or profiling programs can determine the blocking necessary for the best performance. An analytical model for morphological processing memory characteristics is presented which provides for exact cache analysis and prediction. The analytical model is compared to address traces to validate the model. Other algorithms such as inner product, matrix multiplication, and convolution also benefit from the architecture presented herein.

Key Words: Cache tiling, computer architecture, performance, windowed or blocked processing, morphological image processing.

1 Introduction

This paper discusses tiling to achieve high performance while processing images. Small caches with sizes from 2 k to 64 k bytes are implemented on the same chip as the microprocessor. Such on-chip caches do not perform well for windowed image processing if using conventional control. We propose a new architecture to overcome cache conflicts and thrashing (Wittenbrink 1990). We begin by reviewing morphological algorithms and their caching problems. Algorithms used in morphological image analysis work within windows of the image. If not constrained to a window the algorithms work along rows, or columns. The accessing characteristics are algorithm specific but may use memory locations that map to the same places in the cache. Dilation ($\oplus$) and erosion ($\ominus$) are the basic morphological operators (Jain 1989) defined as

$$\mathrm{Dilation}(i, j) = \max\{\mathrm{Image}(i-k, j-l) + \mathrm{SE}(k, l)\} \quad \begin{matrix} (i, j) \in \mathrm{Image} \oplus \mathrm{SE} \\ (k, l) \in \mathrm{SE} \end{matrix}, \qquad \textbf{(EQ 1)}$$

$$\mathrm{Erosion}(i, j) = \min\{\mathrm{Image}(i+k, j+l) - \mathrm{SE}(k, l)\} \quad \begin{matrix} (i, j) \in \mathrm{Image} \ominus \mathrm{SE} \\ (k, l) \in \mathrm{SE} \end{matrix}, \qquad \textbf{(EQ 2)}$$

where (i, j) is a location in the resulting image, and (k, l) is a location of the structuring element denoted by SE. Dilation expands an image and erosions shrinks it. Operations built from erosion and dilation include opening ($\circ$) and closing ($\bullet$). A morphological opening is an erosion followed by a dilation. A closing is a dilation followed by an erosion. Each operation's memory accesses depend on the size and shape of the structuring element. The structuring element used is application dependent. In the example shown in FIGURE 1, (Haralick and Shapiro 1992) the domain of the structuring element is the two points of the input image, I, that the structuring element covers. The structuring element is centered on the left point. The result is $I \oplus \mathrm{SE}$.

Processing with large structuring elements in small caches maps input data to the same locations. This means that previous data are replaced by new data. If the data replaced are to be reused later they must be fetched again. Repeated replacement, called thrashing, is inefficient. Thrashing is one shortcoming of conventional caches for image processing. We propose a mechanism, tiling, to eliminate thrashing for morphological processing. Tiling improves caching when the cache stores the window of interest. The proposed scheme accommodates different structuring elements and image sizes achieving a general solution for windowed processing. Cache tiling improves similar windowed processing such as median filtering, bit block transfers, and matrix operations.

Tiling is used frequently in other applications. Some examples of tiling include Blinn's use of tiling for texture maps (Blinn 1990), and the MPP's and IUA's use of tiling to download image regions to their SIMD arrays (Batcher 1980;Weems 1987). Blinn's application improved the performance of the virtual memory system by avoiding page faults to disk. The SIMD architectures require tiling so that the fixed amount of memory can hold the region of interest for the processing array. Our scheme uses tiling to efficiently cache images. Tiling is efficient because operations are based on near neighbor processing. There are machines that do efficient near neighbor processing, including the MPP (Batcher 1980), the Warwick Pyramid Machine (Nudd et al. 1989), and the IUA (Weems 1987). But these architectures, and fine granularity pipeline processors (Abbot, Haralick and Zuang 1988) such as the Cytocomputer (Loughheed and McCubbrey 1980), cannot process with structuring elements of a large domain, with full color image formats, or gray scale structuring elements. Cache optimization is considered in (Somani et al. 1991) where the difficulty is partially solved with large caches. A general shared memory architecture with tiling can process these formats and achieve the highest performance possible using small caches.

Recent image processing software has been designed to isolate the hardware specifics from the software, but in solutions proposed, software has to explicitly describe the format of the data (Sato et al. 1990), and puts requirements on the hardware for address generation to achieve high performance (Flickner, Lavin, and Das 1990). In the VIEW-Station Environment (Sato et al. 1990), the image format must be changed for the required processing by having the virtual memory controller reformat the data and copy the image to a new location. We avoid reformatting the data, and allow a variety of processing to be done with the highest performance. The control of our architecture is similar to the cache control required of many commercial microprocessors. The image processing software may be used unchanged. The system software sets up the cache depending on the characteristics of the algorithm.

Lam, Rothberg, and Wolf (1990) have recently proposed to use blocking, a concept similar to tiling, for such algorithms. They showed that the cache efficiency is improved with blocking of program loops. Their work relies on the compiler to restructure the loop. However, partitioning the loop results in additional overhead in execution time. Our approach is based on hardware and has minimum overhead in execution time, and combined with blocking can further optimize processing with very large matrices.

In section 2, we discuss tiling and show performance advantages using simulation. In section 3 we discuss implementation and design issues. A tiling trace analysis is done in section 4. An exact analysis is done in section 5 and the trace results are compared with the exact analysis. The paper is concluded in section 6.

2 Tiling

The normal addressing scheme of an image is either by rows or columns. We discuss row major addressing, and the same discussion follows for column major addressing by replacing rows with columns and vice versa. Each pixel in a row is given successive addresses as the row is traversed. Matrices are stored in the same way. An image can be thought of as a matrix of pixel values. Linearization of a 2 dimensional matrix causes elements in the same column, but in consecutive rows to be stored far apart. Depending on the cache size and associativity column elements may map into the same area in the cache. Algorithms that use the columns repeatedly cannot cache them simultaneously. Tiling is addressing an image so that spatial locality creates address locality. This section shows how tiling improves cache utilization while processing images.

2.1 Tiling Principles

Tiling an image breaks the image into a mosaic of windows as shown in FIGURE 2. Addresses in a window are row major, and successive tiles follow in row major order. The addresses within each window are contiguous, the property we exploit for optimal caching in windowed or blocked algorithms. By the notation in FIGURE 2 a $\mathrm{rows} \times \mathrm{cols}$ image consists of $\mathrm{pixels} = \mathrm{rows} \cdot \mathrm{cols}$ elements. The image is addressed by a $(\mathrm{row}, \mathrm{column})$ address. (i, j) denotes any pixel in the image $0 \le i \le \mathrm{rows} - 1$, $0 \le j \le \mathrm{cols} - 1$. A pixel at (i, j) is stored at address $i \cdot \mathrm{cols{+}j}$, and $\mathrm{address} = i \cdot \mathrm{cols{+}j} = \langle i|j \rangle$ is a concatenation of binary strings representing i and j.

The tiled image in the right half of FIGURE 2 is broken into $(\mathrm{rows}/m) \cdot (\mathrm{cols}/n)$ tiles. Tiles are $m \times n$ pixels, with m being the height and n the width of the tile. We assume that rows, cols, m, and n are powers of two. Thus a given pixel (i, j) in an image belongs to tile (r, s), where $r = \lfloor i/m \rfloor$ and $s = \lfloor j/n \rfloor$, and its address in the tile is given by (k, l), where $k = i \,\mathbf{mod}\, m$ and $l = j \,\mathbf{mod}\, n$. The address $\langle i|j \rangle$ can be viewed as consisting of four parts $\langle r|k|s|l \rangle$ where bits representing i are a concatenation of bits representing r and k, and j is a concatenation of s and l. $\langle r|s \rangle$ gives the tile address and $\langle k|l \rangle$ gives the address in the tile. Finally, if an image is stored in a tiled form, then the address in the tiled form is given by $\mathrm{tiled\_address} = r \cdot \mathrm{cols} \cdot m + s \cdot m \cdot n + k \cdot n + l = \langle r|s|k|l \rangle$. To create a tile address from a row major address the k and s bit fields are exchanged,

$$\langle r|k|s|l \rangle \rightarrow \langle r|s|k|l \rangle . \qquad \text{(EQ 3)}$$

2.2 Caching

A cache memory (Stone 1987) is a small fast memory which sits between the processor and the main memory. A cache stores the most commonly used data. It automatically fetches the required data from the main memory and keeps track of data it stores. For this purpose, both cache and main memories are divided into lines of multiple bytes. A cache memory fetches a line at a time from the main memory. The cache may be direct-mapped where each line in the main memory is stored in a specific line in the cache memory. On the other hand in an A-way set-associative cache memory, each line in the main memory may be stored in any one of A banks. An address generated by a processor is split into three fields by the cache as shown in FIGURE 3. The least significant bits, $\mathrm{Word} = \log(\mathrm{bytes/line})$, are used for the word within a line. The remaining two fields are the tag and set address. A cache of size $(\mathrm{cache\_size})$ and associativity (A) has $\mathrm{Set} = \log_2(\mathrm{cache\_size}/A) - \mathrm{Word}$ bits to select the set. The remaining bits, (Tag), are used to establish the correspondence between the main memory line and the line stored in the cache. Further details of cache architecture can be found in Stone (1987).

2.3 Tiling and Caching

Tiling makes a cache memory more efficient. As the processor reads the image it needs only a window of interest to be available in cache memory. In morphology, if the structuring element rows over fill the cache, thrashing occurs as the image is processed, even though there may be sufficient space to store the window of interest. Cache lines are replaced by data in the same window, or domain of the structuring element. The same is true when considering column major addressing of images stored in row major order. We present an example, using a scaled down dilation, to illustrate thrashing. The same problem occurs with typical images and current on-chip cache sizes. Consider dilation of an 8 pixel wide image, with a 2x4 structuring element: m=2, n=4, rows = 8, and cols = 8. The structuring element is used like a convolution mask. Let (k_0, l_0) be the center of the structuring element, in this example $(k_0 = 1, l_0 = 3)$. The algorithm is shown in FIGURE 4.

The cache has 8 lines and is direct mapped. The behavior of the cache for computing results 0 to 4 is shown in FIGURE 5. The lines are denoted as line 0, line 1 etc., and pixels from the input image are shown stored in the cache as *p0*, *p1*, . . . , *p15*. As the window is covered the cache reads pixels in and replaces each previously read pixel, not using half of the cache. Notice that each pixel read in after pixel 3 replaces one already in the cache. The processing in the 2x4 window of interest uses these pixels again, and they have to be repeatedly fetched from main memory. While having a direct mapped cache exacerbates the problem shown above, the same effect will occur in 2 or 4 way set associative caches when the number of rows exceeds the number of sets for that address.

By tiling the previous example, with a 2x4 tile we can make better use of all 8 lines of the cache. The algorithm is the *same* as shown in FIGURE 4. Now the *cache* transforms the addresses. The image in main memory is stored the same, but tiling occurs when caching. An image, its tiled form, and cache replacement are shown in FIGURE 6. The tile address is calculated by using the address reshuffling formula (EQ 3) as shown in FIGURE 7. Computing results 0 to 4, without tiling, there are 40 reads from main memory (FIGURE 5). To compute results 0 to 4, with tiling, there are 16 reads (FIGURE 6). The number of reads to compute all results in this example are 390 without tiling and 120 with tiling.

To make caching more efficient without tiling we can reuse pixels by alternating the traverse order of k and l for reading the structuring element. Such snake like processing (Hwang and Briggs 1984) still requires 28 reads to compute results 0 to 4. The pixels most recently read in are reused for the following loop. In FIGURE 5, after 8 pixels have been read for the first result, the order of reading for the second result is *p12* (miss), *p11* (hit), *p10* (hit), and *p9* (hit). Then *p1*, *p2*, *p3*, and *p4* (misses) are read. This saves misses on reads of *p9*, *p10*, and *p11*. But, the software overhead grows, and compilers and replacement policies of the cache may defeat coding intentions. Because the algorithm must be changed and tiling gives better results we do not use snake like processing. Snake like processing with tiling is briefly discussed later in section 5.

3 Cache Tiling Design

Tiling can be done in software by storing the image in tiled form in main memory. But, software tiling is not the best solution. In fact, there are four important advantages to storing images untiled in main memory. First, the image storing and transferral mechanisms, such as DMA or I/O devices, are independent of the tiling scheme. Second, different tilings may be used on the same image. Third, the same tile windows can be used to process arbitrary sized images. And the fourth, and possibly most important, advantage is that the address translation is transparent to user software. These advantages dictate that a conversion of the address take place when caching, and not in software. Hardware tiling improves performance. System software calls are made by the users to set up the cache for higher performance. For example, to allocate tiled memory a call to a custom allocation routine can set up the memory area to be tiled while cached.

The tiling example shown in the previous section requires a fast address translator to exchange the k and s fields as shown in (EQ 3). For general use, the translator should be able to implement a set of tilings that may be used with any algorithm of interest. By minimizing tilings and image formats, a reasonable translator is designed. The cache translates the address for all reads and writes to determine a hit or miss as shown in FIGURE 8. Because the translation is done for all accesses the operation is time critical. The mode information regarding the size of the image and tile is stored in a programmable register, selected by address bits, or activated by the memory management unit.

3.1 Design Methodology

In image processing, several image frames may be processed together to compute a result. Let J be the number of windows from different image frames required simultaneously for processing. To avoid thrashing, the cache should be partitioned to hold J windows at once. If the cache is A-way set associative, $A > J$, and each cache memory bank is large enough to hold a tile, then this is achieved automatically. Otherwise, the cache set addresses are split in $P = J/A$ partitions. In this case, a memory address consists of 6 fields as shown in FIGURE 9. Here, field q represents the upper bits in the address which are not used for tiling. Field p includes the bits representing the number of partitions and r, s, k, and l are as defined earlier. To partition the cache to accommodate tiles from all images, p directs images to different sets in the cache. Care must be taken to use images with different partition addresses (bits in the p field), or the images will collide defeating the purpose of the mapping.

An image processing system computes with different image and window sizes. Each set of specifications is a mode, denoted by ϕ. Let $(\mathrm{rows}_\phi \times \mathrm{cols}_\phi)$ be the image size, J_ϕ be the number of images, and $(m_\phi \times n_\phi)$ be the size of tile window required for mode ϕ. If $m_\phi \cdot n_\phi \cdot J_\phi$ is smaller than the $(\mathrm{cache\_size})$ then all images can be tiled and cached simultaneously. Combined software blocking (Lam, Rothberg, and Wolf 1990) with hardware tiling can allow even larger tiles to be used by reducing the $(m_\phi \times n_\phi)$ required for the algorithm. We will discuss only hardware tiling and design. Tiling improves cache performance when the number of image rows that can be cached is less than the rows in the window of interest, or $\mathrm{cache\_size} < m_\phi \cdot \mathrm{cols}_\phi \cdot J_\phi$. For any mode, tiling is useful and possible if

$$m_\phi \cdot n_\phi \cdot J_\phi \le \mathrm{cache\_size} < m_\phi \cdot \mathrm{cols}_\phi \cdot J_\phi, \qquad \phi \in \mathrm{Modes}\,. \tag{EQ 4}$$

When (EQ 4) is satisfied for all modes, we create a tile translator by mapping mode parameters into design parameters, $\phi \to d\phi$, and derive a circuit. The number of partitions $(P_{d\phi})$ is determined from J_ϕ and the associativity (A) of the cache,

$$P_{d\phi} = \lceil J_\phi / A \rceil, \qquad J_\phi \ge A\,. \tag{EQ 5}$$

When $J_{\phi} < A$, $P_{d\phi} = 1$ minimizes the required tiling. The given image size dictates that $\mathrm{rows}_{d\phi} = \mathrm{rows}_{\phi}$ and $\mathrm{cols}_{d\phi} = \mathrm{cols}_{\phi}$. We partition the cache into storage for each image that will be simultaneously in the cache. The cache associativity and the partition bits divide up the cache. Tile dimensions $m_{d\phi}$ and $n_{d\phi}$ are chosen by tessellating the partition of size

$$\mathrm{p\_size} = \mathrm{cache\_size} / (P_{d\phi} \cdot A) \ . \tag{EQ 6}$$

Width $n_{d\phi}$ is at least as large as the line size of the cache or $n_{d\phi} \geq \mathrm{line\_size}$. Also, $m_{d\phi}$ and $n_{d\phi}$ are chosen in such a way that $m_{d\phi} \geq m_{\phi}$, $n_{d\phi} \geq n_{\phi}$, and

$$m_{d\phi} \times n_{d\phi} = \mathrm{p\_size} \ . \tag{EQ 7}$$

A user can adjust $m_{d\phi}$ and $n_{d\phi}$ to reduce the overall cost as long as $m_{d\phi} \times n_{d\phi} \leq \mathrm{p\_size}$. Width $n_{d\phi}$ should be maximized if the processing is being done in the row major order, and $m_{d\phi}$ should be maximized if the processing is being done in the column major order. All these parameters have to be powers of 2 for these formula and for efficient implementation. If $n_{d\phi}$ is greater than or equal to the width of the image ($\mathrm{cols}_{d\phi}$) then no tessellation is required.

Each mode requires address translation to perform partitioning and tiling of FIGURE 8 determined by $P_{d\phi}$, $m_{d\phi}$, $n_{d\phi}$, and $\mathrm{rows}_{d\phi} \times \mathrm{cols}_{d\phi}$. A reduction of modes is done by removing those covered by the original cache, and equivalent modes. By multiplexing the input address bits to the output address bits for all of the modes, $d\phi$, and regular caching a $\Phi + 1$ multiplexor directly implements the circuit, where Φ is the number of modes. The range of bits that are translated is computed using (EQ 8), (EQ 9), (EQ 10), (EQ 11), and (EQ 12). Different field lengths are computed using the following relations.

$$(b_p)_{d\phi} = \log_2 P_{d\phi} \qquad (b_r)_{d\phi} = \log_2 \mathrm{rows}_{d\phi} - \log_2 m_{d\phi} \qquad (b_s)_{d\phi} = \log_2 \mathrm{cols}_{d\phi} - \log_2 n_{d\phi}$$
$$(b_k)_{d\phi} = \log_2 m_{d\phi} \qquad (b_l)_{d\phi} = \log_2 n_{d\phi} \ . \tag{EQ 8}$$

The least significant bit of the translator is

$$\mathrm{b\_right} = \min \{ (b_l)_{d\phi} \} , \qquad d\phi \in \mathrm{Modes} \ . \tag{EQ 9}$$

When no partitioning is required, the most significant bit of any tiling mode is

$$\mathrm{b\_left} = \max \{ ((b_l)_{d\phi} + (b_k)_{d\phi} + (b_s)_{d\phi}) - 1 \} , \qquad d\phi \in \mathrm{Modes} \ . \tag{EQ 10}$$

Otherwise, the most significant bit of a combination of tiling and partitioning is

$$\text{b\_left\_p} = \max\{((b_l)_{d\phi} + (b_k)_{d\phi} + (b_s)_{d\phi} + (b_r)_{d\phi} + (b_p)_{d\phi}) - 1\}, \quad d\phi \in \text{Modes}, P_{d\phi} > 1 \quad . \quad \textbf{(EQ 11)}$$

It follows that the width of the translator required is

$$\#(\text{bits}) = \max\{\text{b\_left}, \text{b\_left\_p}\} - \text{b\_right} + 1 \ . \quad \textbf{(EQ 12)}$$

In the next section we present detailed examples.

3.2 Cache Address Translator Example

We present translators with and without partitioning. Suppose we are interested in image sizes 256x256, 512x512, 1024x1024, and structuring elements 32x1 and 1x32. The cache we are using is 8 K two way set associative ($A = 2$) with a line size of 32 bytes. Tiles can be any size as long as they cover the required window. To hold one image, $J = 1$. Using equations (EQ 4) to (EQ 12), the design parameters are $P_{d\phi} = 1$, $m_{d\phi} \times n_{d\phi} = \{64 \times 64\}$ for all image sizes. The field widths computed using the relations in (EQ 8) are given in TABLE 1 in the $P_{d\phi} = 1$ column. The translator locations are: $\text{b\_right} = 6$, $\text{b\_left} = 15$ and $\#(\text{bits}) = 10$.

If we require many images to be simultaneously cached, say $J = 8$, then $P_{d\phi} = 4$ and $m_{d\phi} \times n_{d\phi} = \{32 \times 32\}$ for all image sizes. In this case field widths are given in TABLE 1 in the $P_{d\phi} = 4$ column. The bit locations are $\text{b\_right} = 5$, $\text{b\_left\_p} = 21$ and $\#(\text{bits}) = 17$. A circuit to implement either translation is shown in FIGURE 10. The translator in each case consists of 4 to 1 multiplexors for each output. It uses only two levels of logic for high speed.

TABLE 1 Cache Address Translator Tilings

	$P_{d\phi} = 1$ 64×64				$P_{d\phi} = 4$ 32×32				
image size	$\mathbf{b_r}$	$\mathbf{b_k}$	$\mathbf{b_s}$	$\mathbf{b_l}$	$\mathbf{b_p}$	$\mathbf{b_r}$	$\mathbf{b_k}$	$\mathbf{b_s}$	$\mathbf{b_l}$
256x256	2	6	2	6	2	3	5	3	5
512x512	3	6	3	6	2	4	5	4	5
1024x1024	4	6	4	6	2	5	5	5	5

Implementations that tile into the line size ($n_\phi < \text{line\_size}$) require advanced control to perform fills and flushes. Tiling by at least the line size allows for simpler control. A single line's data are fetched from contiguous addresses. The same translator may be used for different cache sizes by reducing the tile width, n, and shifting the translator or reducing b_right. The next section discusses a dilation address trace analysis that illustrates the performance improvement from cache tiling.

4 Tiling Trace Analysis

A good performance estimate for comparing cache architectures is the average cycle time (Stone 1987). Total time for an application is roughly the sum of the number of cycles spent executing and the cycles spent waiting for the memory system (Hennessy and Patterson 1990). The majority of processor stalls result from cache misses. The hit ratio, or percentage of accesses satisfied in the cache, can be used as an implementation independent measure of cache performance. The cycle time depends upon the technology used, VLSI line spacing, process, and other production factors. The hit ratio depends only upon the cache behavior and the program behavior. Therefore, an accepted technique for cache performance comparisons is to measure the hit ratio. We developed a simulator that processes address traces captured on a workstation to compare hit ratios with and without cache tiling.

Dilation using a structuring element of 32 rows and one column, or a rod, was used to generate the address trace. We used tiles of size $32 \times n_{d\phi}$ by extending $n_{d\phi}$ as much as required. Dilations of image sizes 256x256, 512x512, and 1024x1024 were traced. 2 way set associative caches from 2 kilobytes to 1 Megabyte were used to process the traces. The results are given in FIGURE 11 and FIGURE 12. See also TABLE 2, TABLE 3, and TABLE 4. The trace is of all data addresses generated while executing the dilation. Erosion, opening, and closing operations will give similar results as will be shown in the analytical analysis section. The trace results are not dependent on the data, only upon the size of the image and the structuring element.

FIGURE 11 shows the hit ratios for the input of the 256x256 image with four cache sizes from 2 k bytes to 16 k bytes. Caches larger than 16 k do not further improve the hit ratios. Only the input image is cached. The structuring element is not cached. The results show that a 2 k byte regular cache gives a hit ratio of zero, while the tiling cache gives a hit ratio of 0.96875. This shows the strength of tiling. When the regular cache reaches a size of 8k bytes, it has enough capacity to store all the rows of interest in the image, so at this threshold tiling is no longer needed. This will be shown analytically in section 5. FIGURE 12 shows the effect of tiling using all three image sizes. The cache size necessary to achieve a high hit ratio without tiling doubles as the image width doubles. Tiling is nearly optimal for all sizes and formats.

5 Exact Cache Analysis

Dilation and erosion are regular and predictable and an analytical analysis is done to determine the cache hit ratio for ideal cache behavior. This is done to show that tiling approximates ideal cache behavior for the algorithms of interest and is therefore optimal. Dilation and erosion are used to determine closed form solution of hit ratios assuming grey scale structuring elements. Such structuring elements do not allow row and column decomposition of dilation and erosion. Thus the overall computation is more demanding. The calculation requires the explicit use of all members of the structuring element. These assumptions are valid for grey scale operations with large structuring elements that cannot be stored in registers. To interpret the results for processing where the structuring elements can be held in the registers, and for processing that does not iterate through the structuring element, can easily be done by removing the number of structuring element reads from the analysis. A structuring element is centered at $(k = k0, l = l0)$, where indexing of the structuring element is $k = 1 \text{to} m$, and $l = 1 \text{to} n$. The center affects the number of reads and cache misses for dilation and erosion when the input, or the output is limited. We consider erosion first and then dilation for the number of reads, writes, and cache misses.

Two different sizes of cache are used. The caches, to simplify the derivations, use OPT, or optimal replacement (Stone 1987), and are fully associative. In this way dependencies on memory alignment, and order of replacement do not affect the analytical analysis, and the cache may perform no better than OPT. Cache 1 holds the input window, $m \cdot n$ pixels, and the structuring element, $m \cdot n$, or the $\text{cache\_size} = 2 \cdot m \cdot n$. The second cache, Cache 2 is of $\text{cache\_size} = mC + mn$, where C is the number of columns in the input. The number of columns changes for each variety of algorithm as discussed later. This cache holds all rows of interest and optimally caches the input. Cache 1 is a fully associative cache that holds the window of interest. Tiling caches approximate Cache 1. Cache 2 is the best performance that can be expected with any cache. Cache 1 and 2 have b bytes per cache line. The units of m, n, and C are in pixels. If a pixel does not equal a byte then the bytes per line factor, b, should contain a pixel to byte conversion, $b\,(\text{pixel/byte})\,(\text{byte/line})$, for computation of misses. This section concludes by directly comparing the analytical models to the trace analysis done in Section 4.

5.1 Erosion

To derive the number of reads and writes we examine the input and result of erosion. For grey scale erosion the input is $\mathrm{rows} \times \mathrm{cols}$, and the result, $(\mathrm{rows} - m + 1) \times (\mathrm{cols} - n + 1)$, is smaller than the input. The result's coordinate frame is translated relative to the original input by $k0 - 1, l0 - 1$. The images used for erosion are shown below in FIGURE 13.

For each pixel result, we read $m \cdot n$ input pixels and $m \cdot n$ structuring element values. The number of reads and writes for the erosion are

$$\#(\mathrm{reads}) = 2(\mathrm{rows} - m + 1)(\mathrm{cols} - n + 1)m \cdot n , \tag{EQ 13}$$

$$\#(\mathrm{writes}) = (\mathrm{rows} - m + 1)(\mathrm{cols} - n + 1) . \tag{EQ 14}$$

Assuming that the image is processed in row order, we need to read a $m \times \mathrm{cols}$ window each time one row of output is processed. To read the structuring element, we need to perform $(mn)/b$ reads. The number of bytes per cache line, given by b, helps to significantly reduce the number of misses. The number of read misses for Cache 1 is

$$\mathrm{Cache\ 1,}\ \#(\mathrm{misses}) = (\mathrm{rows} - m + 1)\,\mathrm{cols} \cdot \frac{m}{b} + \frac{mn}{b} . \tag{EQ 15}$$

Cache 2 is large enough to hold all rows in the window of interest. Thus the entire image is read only once. The width of the input is $(m = \mathrm{cols})$. The number of read misses is

$$\mathrm{Cache\ 2,}\ \#(\mathrm{misses}) = \mathrm{rows} \cdot \frac{\mathrm{cols}}{b} + \frac{mn}{b} . \tag{EQ 16}$$

To calculate the hit ratio using these equations is straight forward. We consider writes as misses, and the number of hits as: the total number of reads, minus, the number of cache read misses. For both Cache 1 and Cache 2 the hit ratio is

$$\mathrm{hit\ ratio} = \frac{\#(\mathrm{reads}) - \#(\mathrm{misses})}{\#(\mathrm{reads}) + \#(\mathrm{writes})} . \tag{EQ 17}$$

5.2 Dilation

Using the same derivation as for the erosion, the reads, writes, and misses for dilation is calculated. For dilation a $\mathrm{rows} \times \mathrm{cols}$ image processed with a $m \times n$ structuring element requires as input an enlarged $(\mathrm{rows} + 2m - 2) \times (\mathrm{cols} + 2n - 2)$ image. The result is larger than the original input, and has a size of $(\mathrm{rows} + m - 1) \times (\mathrm{cols} + n - 1)$. The expanded input and output images are shown in FIGURE 14. The expanded image can be created by inserting the minimum pixel value before dilation, or the edges of the original image may be extended. The expanded input image coordinates are shifted in relation to the original image by $(-m+1, -n+1)$. The number of reads, writes, and misses are ($C = \text{cols} + 2n - 2$):

$$\#(\text{reads}) = 2\,(\text{rows} + m - 1)\,(\text{cols} + n - 1)\,m \cdot n\,, \tag{EQ 18}$$

$$\#(\text{writes}) = (\text{rows} + m - 1)\,(\text{cols} + n - 1)\,, \tag{EQ 19}$$

$$\text{Cache 1,}\;\; \#(\text{misses}) = (\text{rows} + m - 1)\,(\text{cols} + 2n - 2)\,\frac{m}{b} + \frac{mn}{b}\,, \tag{EQ 20}$$

$$\text{Cache 2,}\;\; \#(\text{misses}) = \frac{(\text{rows} + 2m - 2)\,(\text{cols} + 2n - 2)}{b} + \frac{mn}{b}\,. \tag{EQ 21}$$

As an example, FIGURE 15 depicts dilation of a 4x4 image by a 2x3 structuring element with *k0*=2, *l0*=3. FIGURE 15 (A), shows the reads of the input image only. The number of reads is twice that amount when the structuring element reads are added. FIGURE 15 (B) shows the number of misses for Cache 1, and FIGURE 15 (C) shows the number of misses for Cache 2. To perfectly cache the image means only having to read the image once. Because the line size $b = 1$, every pixel must be read. (EQ 18), (EQ 19), (EQ 20), and (EQ 21) are used to calculate the reads, writes and misses. (EQ 17) is used to calculate the hit ratio, which for this example is 0.702564 for Cache 1 and 0.784615 for Cache 2.

5.3 Dilation With Restricted Input and Output

For memory limited applications the dilation can be limited to produce an output of only $\text{rows} \times \text{cols}$, and not the $(\text{rows} + m - 1) \times (\text{cols} + n - 1)$ image. The dilation operates on a $\text{rows} \times \text{cols}$ image and creates a $\text{rows} \times \text{cols}$ image. We consider processing in this manner and derive optimal hit ratios for dilation. Both $\text{rows} \times \text{cols}$ input and extended $(\text{rows} + m - 1) \times (\text{cols} + n - 1)$ input cases are considered as shown in FIGURE 16. The input images considered both produce a $\text{rows} \times \text{cols}$ output image. The output is in the same position as the input. The expanded input has its coordinates translated by $k0 - m, l0 - n$ in relation to the original input.

The number of reads of input data may be calculated by summing the number of reads for each pixel. We derive the number of reads, and the number of misses in the cache to calculate the cache hit ratios. The number of reads is a 2 dimensional histogram that returns the number of reads for any pixel in the enlarged input image for dilation to produce a $\text{rows} \times \text{cols}$ image. The histogram of reads is given by

$$F(i,j) = \begin{cases} i \cdot f(j) & 1 \le i \le m-1 \\ m \cdot f(j) & m \le i \le \text{rows} \\ [(\text{rows}+m)-i] \cdot f(j) & \text{rows}+1 \le i \le \text{rows}+m-1 \end{cases}$$

$$f(j) = \begin{cases} j & 1 \le j \le n-1 \\ n & n \le j \le \text{cols} \\ (\text{cols}+m)-j & \text{cols}+1 \le j \le \text{cols}+n-1 \end{cases} \quad . \qquad \text{(EQ 22)}$$

If we consider all of the reads in the extended image and structuring element reads we have

$$\#(\text{reads}) = 2 \sum_{i=1}^{\text{rows}+m-1} \sum_{j=1}^{\text{cols}+n-1} F(i,j) = 2 \cdot \text{rows} \cdot \text{cols} \cdot m \cdot n \ . \qquad \text{(EQ 23)}$$

If we want only the reads that fall within the original $\text{rows} \times \text{cols}$ image, as our conditional dilation used in sections 2.3 and 4, we limit the range of $F(i,j)$. For structuring elements that are centered at $(k = k0, l = l0)$ consider the general form

$$\#(\text{reads}) = 2 \sum_{i=m-k0+1}^{\text{rows}+m-k0} \sum_{j=n-l0+1}^{\text{cols}+n-l0} F(i,j) \, . \qquad \text{(EQ 24)}$$

If the structuring element is centered in the upper left hand corner ($k0 = 1$, $l0 = 1$), or the lower right hand corner ($k0 = m$, $l0 = n$),the number of reads in the image and in the structuring element is

$$\#(\text{reads}) = 2\sum_{i=1}^{\text{rows}} \sum_{j=1}^{\text{cols}} F(i,j) = 2\left(\sum_{i=1}^{m-1} \sum_{j=1}^{n-1} ij + \sum_{i=1}^{m-1} \sum_{j=n}^{\text{cols}} in + \sum_{i=m}^{\text{rows}} \sum_{j=1}^{n-1} jm + \sum_{i=m}^{\text{rows}} \sum_{j=n}^{\text{cols}} mn \right)$$

$$= 2\left(\text{rows} - \frac{m}{2} + \frac{1}{2}\right) \cdot \left(\text{cols} - \frac{n}{2} + \frac{1}{2}\right) \cdot m \cdot n \quad . \qquad \text{(EQ 25)}$$

(EQ 24) and (EQ 25) give the number of reads of pixels in the input image and number of reads of the structuring element. There are closed form solutions for each of the ranges so that four solutions can be derived from (EQ 24). The ranges are: $k0 = 1 \text{ or } m$, $l0 = 1 \text{ or } n$ (EQ 25); $1 < k0 < m$, $l0 = 1 \text{ or } n$; $k0 = 1 \text{ or } m$, $1 < l0 < n$;$1 < k0 < m$, $1 < l0 < n$. To dilate an image with a dense structuring element requires that many reads. The number of writes for the dilation is independent of the form of the input. For both expanded input, and the $\text{rows} \times \text{cols}$ input the number of writes is

$$\#(\text{writes}) = \text{rows} \cdot \text{cols} \ , \qquad \text{(EQ 26)}$$

because we have chosen to limit the size of the result to the size of the input.

Next we derive the number of misses for Cache 1 using both input cases shown in FIGURE 16. The number of misses depends upon both the initial structuring element reads and the input image reads. For processing the extended input image $(\mathrm{rows} + m - 1) \times (\mathrm{cols} + n - 1)$ the number of cache read misses while processing with the cache of $2 \cdot m \cdot n$ is

$$\text{Cache 1, } \#(\text{misses}) = \mathrm{rows} \cdot m \cdot \frac{(\mathrm{cols} + n - 1)}{b} + \frac{m \cdot n}{b} . \qquad \textbf{(EQ 27)}$$

For the $\mathrm{rows} \times \mathrm{cols}$ input the number of misses is

$$\text{Cache 1, } \#(\text{misses}) = \left(\mathrm{rows} \cdot m - \frac{1}{2}k0\,(k0 - 1) - \frac{1}{2}(m - k0 + 1)\,(m - k0)\right)\frac{\mathrm{cols}}{b} + \frac{m \cdot n}{b} . \qquad \textbf{(EQ 28)}$$

Cache 2 holds all rows of interest. $C = \mathrm{cols} + n - 1$ for the extended input, and $C = \mathrm{cols}$ for the reduced input. The number of read misses in Cache 2 for the two input variations is

$$\text{Cache 2, } \#(\text{misses}) = \frac{(\mathrm{rows} + m - 1)\,(\mathrm{cols} + n - 1)}{b} + \frac{mn}{b} , \qquad \textbf{(EQ 29)}$$

$$\text{Cache 2, } \#(\text{misses}) = \frac{\mathrm{rows} \cdot \mathrm{cols}}{b} + \frac{mn}{b} , \qquad \textbf{(EQ 30)}$$

for extended input images, and reduced input respectively.

Again, the hit ratios are computed by using (EQ 17). In FIGURE 17 is an example showing the number of reads (FIGURE 17 A) and misses for Cache 1 (FIGURE 17 B) and Cache 2 (FIGURE 17 C). The structuring element is centered at $(k0 = m,\ l0 = n)$. The number of image reads, and not structuring element reads, or result writes is shown to simplify the example. Hit ratios are calculated from (EQ 25), (EQ 26), (EQ 28), (EQ 30), and (EQ 17), and the ratios are 0.647887 for Cache 1 and 0.732394 for Cache 2.

5.4 Snake-Like Processing for Reducing Misses

When the cache is large enough to hold the window of interest to calculate 1 result the input is read only once to compute a result (Cache 1). The input will be read again when the rows are traversed for following results. We may save a small number of misses if we change the direction of processing on each row. This is called snake-like row major processing because the coverage path of the input winds (Hwang and Briggs 1984). Normal processing as shown by the algorithm in section 2.3 and snake-like processing are shown in FIGURE 18.

Snake-like processing for small examples appears to be quite beneficial, but processing typical sized images the savings are small and the programming is more difficult. The number of misses saved for the reduced $\mathrm{rows} \times \mathrm{cols}$ input to produce a reduced $\mathrm{rows} \times \mathrm{cols}$ output is given by

$$\#(\text{misses saved}) = (\text{rows} - 1)\,(m - 1)\frac{n}{b}\,. \qquad \textbf{(EQ 31)}$$

The savings for the dilation in section 4 would have been only 7,905 misses or 12% using $m = n = 32$ and $b = 32$, because the structuring element has to be in units of cache lines or 32 bytes. The hit ratio would be improved from 0.96875 to 0.9726. In addition to snake-like processing by rows the structuring element may be traversed in this way so that values are re-used as much as possible. By not using snake-like processing to evaluate the number of reads we are presenting a worst case estimate of the number of reads which is consistent with avoiding compiler and programming dependence.

5.5 Analytical Analysis and Trace Analysis Comparison

To examine how the analytical results compare to practice, we consider the statistics from the dilation traces, presented in Section 4. First, we consider the impact of input image reads. The results are shown in TABLE 2 and TABLE 3. The trace driven simulation matches the analytical results for the appropriate cache sizes. Both the trace (for 2 k and 4 k tiling caches) and (EQ 28) (using only the term for input image reads) give 63488 misses. Because we have the same number of misses for the tiling caches the hit ratios are the same also. From (EQ 17) both have hit ratios of 0.96875.

TABLE 2 256x256 Dilation Analytical and Trace Hit Ratios, Input Only

	2 k	4 k	8 k	16 k	32 k
Regular Trace	0.0	0.01406	0.94040	0.99899	0.99899
Tiling Trace	0.96875	0.96875	0.99749	0.99899	0.99899
Cache 1 (2k)	0.96875	0.96875	-	-	-
Cache 2 (8K+32)	-	-	0.99899	0.99899	0.99899

TABLE 3 256x256 Dilation #(misses), Input Only

	2 k	4 k	8 k	16 k	32 k
Regular Trace	2031616	2003056	121088	2048	2048
Tiling Trace	63488	63488	5108	2048	2048
Cache 1 (2k)	63488	63488	-	-	-
Cache 2 (8K+32)	-	-	2048	2048	2048

The reason the regular cache does not match Cache 1 is that the mapping thrashes, as discussed in Section 1. Tiling gives the same results because tiling approximates the full associativity used in the analytical models. The second cache model, Cache 2, is duplicated for the caches that optimally cache the input. When considering only the input reads the trace exactly matches the analytical results. Caches greater than 16 k, tiling or regular, gives 2048 misses, 2,029,568 hits, and a hit ratio of 0.99899 for input reads only. The tiling cache of 8k is slightly less efficient than optimal because the order of processing for the algorithm returns to the image edges after the middle has been read, because loop unrolling is used to optimize the algorithm. Equations for the second cache model give 2048 misses, (EQ 30), 2,031,616 reads, and also a hit ratio of 0.99899.

The complete traces including structuring element reads and image writes show how the model matches the trace. Image reads, structuring element reads, and result writes are used for hit ratios given in TABLE 4. Considering structuring element reads and result writes the hit ratios are calculated from (EQ 17). The analytical cache model, Cache 1**,** and the 4 k tiling cache trace analysis both yield a hit ratio of 0.96875. The 2 k tiling cache yielded 0.937912, and does not match with the analytical result because the structuring element gets replaced several times as it collides with the input image. In addition, the order of processing in the input image returns to the edges as mentioned before. The 8 k tiling cache was nearly optimal, 0.98340 versus 0.98363, but doesn't have storage for the 32 byte structuring element that misses. Cache 2 yields a hit ratio of 0.983631. From the trace analysis all caches greater than 16 k, tiling or regular give hit ratios of 0.983630. A cache size of 16 k yields the optimal hit ratio predicted by Cache 2. The cache size to yield the optimal hit ratio is 8k (rows of interest) + 32 (structuring element size).

TABLE 4 256x256 Dilation Analytical and Trace Hit Ratios

	2 k	**4 k**	**8 k**	**16 k**	**32 k**
Regular Trace	0.48827	0.49769	0.97516	0.98363	0.98363
Tiling Trace	0.93791	0.96875	0.98340	0.98363	0.98363
Cache 1 (2k)	0.96875	0.96875	-	-	-
Cache 2 (8K+32)	-	-	0.98363	0.98363	0.98363

The analytical results predict the trace analysis. They differ in cases where the cache cannot approximate a fully associative cache, and overlaps structuring element and image, or the processing does not truly process the entire image in increasing row or column order. The read only trace shows that the first cache model is valid for tiling caches, and not regular caches. This shows how tiling helps to approximate full associativity. When cache is large enough for all rows Cache 2 predicts performance for both tiling and regular caches.

6 Conclusions

Tiling improves hit ratios of small caches as shown through analysis. Optimal caching occurs when the cache is large enough to store all rows of the image and the structuring element being used in processing. The data are read exactly once. Windowed processing allows near optimal caching with much less storage. Windowed processing is advantageous when the data structures being processed are large, and the window of processing is not. We showed how to achieve ideal windowed caching with direct mapped and associative caches through cache tiling. Trace analysis duplicated the hit ratios of derived analysis confirming the analytical model and showed tiling matches the fully associative cache that holds the window. A larger percentage of hits speeds up processing. Both tiling and partitioning allow smaller size on chip caches to achieve the same performance as larger external caches. The implementation is straight forward, and can be minimized for specific requirements to reduce chip area. Tiling remains useful for on chip cache control because as cache sizes increase image sizes will also increase. Tiling improves caching for any image or matrix processing that reuses data. The same improvements as shown for dilation will improve other morphological and matrix algorithms.

For example, in calculating matrix product, $C = AB$, matrices must be accessed by row (A) and by column (B). One way to make this efficient for memory accesses is to transpose (B) before the multiplication. This requires reformatting of the data, similar to software tiling. This allows data that are fetched to a line to all be used. Otherwise, the column traversed matrix will miss on every read. The row traversed matrix will miss on each new line, but all elements within a line will be hits. Depending upon the size of the matrix and the size of the cache, the decision as to which values are to be cached can be (A) and/or (B) and/or (C). One alternative, by caching (A) and (B) we can avoid caching writes of (C). Using cache tiling, matrix (B) is tiled to maximize the number of columns being held in the cache. The cache remaps the addresses to store the data by columns. Because we tile by at least the line size, the $\mathrm{rows} \times \mathrm{line\_size}$ elements stored in the cache will all be saved, and following column reads of (B) will hit. Thus, cache tiling avoids the reformatting step, allows natural indexing in the user program, and provides highly efficient memory system performance. With larger matrices where an entire row cannot be stored in the cache, we can use block processing, and tiling will allow the blocks to be stored in the cache without reformatting. Other image and windowed processing, where the cache cannot store the entire data structure, but may store the window of processing (in this example a column or block) can use tiling to achieve fully associative performance with direct mapped or set associative caches.

Higher cache efficiency improves MIMD shared memory performance for image processing. Furthermore, tiling affects only the private copy of the data thereby not impacting other processors, shared memory, or DMA devices. Bus snoopers and other coherency protocols are independent of the tiling, but must use the same mapping as the tiling translator to compare tags. Tiling is a method of increasing the apparent associativity of the cache for matrices and images. Further research may show how tiling with a direct mapped cache removes dependence upon order of replacement, a weakness of set associative caches. Compiler optimization for use of variables can be defeated if the cache replacement policy is not deterministic. We will also do analysis of other algorithms using tiling caches.

References

Abbott L, Haralick RM, Zhuang X (1988) Pipeline Architectures for Morphological Image Analysis. Machine Vision and Applications 1:23–40

Batcher KE (1980) Design of a Massively Parallel Processor. IEEE Transactions on Computers 29(9):836–840

Blinn JF (1990) Jim Blinn's Corner "The Truth About Texture Mapping." IEEE Computer Graphics and Applications :78–83,March

Flickner M, Lavin M, Das S (1990) An Object-oriented Language for Image and Vision Execution (OLIVE). in 10th Annual IEEE International Conference on Pattern Recognition, June

Haralick RM, Shapiro LG (1992), Computer and Robot Vision Vol. 1. Addison-Wesley, Reading, MA, pp 157–262

Hennessy JL, Patterson DA (1990), Computer Architecture A Quantitative Approach. Morgan Kaufmann, San Mateo, CA, pp 416 –419

Hwang K, Briggs FA (1984), Computer Architecture and Parallel Processing. McGraw Hill, pp 362-363

Jain AK (1989) Fundamentals of Digital Image Processing. Prentice Hall, Englewood Cliffs, NJ

Lam M, Rothberg EE, Wolf ME (1990) The Cache Performance and Optimizations of Blocked Algorithms. in Proceedings of the ASPLOS-IV, pp 63–74

Loughheed RM, McCubbrey DL (1980) The Cytocomputer: A Practical Pipelined Image Processor. in Proceedings of the 7th Annual Symposium on Computer Architecture, pp 271–277

Nudd GR, Atherton TJ, Francis ND, Howarth RM, Kerbyson DJ, Packwood RA, Vaudin GJ (1989) WPM: A multiple-SIMD architecture for image processing. in Proceedings Third International IEE Conference on Image Processing and Its Applications, London, pp 161–165, July

Sato H, Okazaki H, Kawai T, Yamamoto H, Tamura H (1990) The VIEW-Station Environment: Tools and Architecture for a Platform-Independent Image-Processing Workstation. in 10th Annual IEEE International Conference on Pattern Recognition, pp 576–583, June

Somani AK, Wittenbrink CM, Haralick RM, Shapiro LG, Hwang JN, Chen CH, Johnson R, Cooper K (1991) Proteus System Architecture and Organization. in Proceedings of Fifth International Parallel Processing Symposium, pp 276–284, April

Stone HS (1987) High Performance Computer Architecture. Addison Wesley, Reading, MA

Weems CC, Levitan SP, Hanson AR, Riseman EM, Nash JG, Shu DB (1987) The Image Understanding Architecture. COINS Technical Report 87–76, University of Massachusetts

Wittenbrink CM (1990) Directed Data Cache for High Performance Morphological Image Processing. Master's thesis, Dept. of Elect. Engr., University of Washington, Seattle, WA

Figures

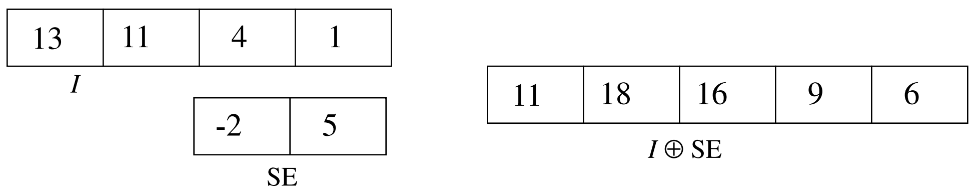

FIGURE 1 Dilation

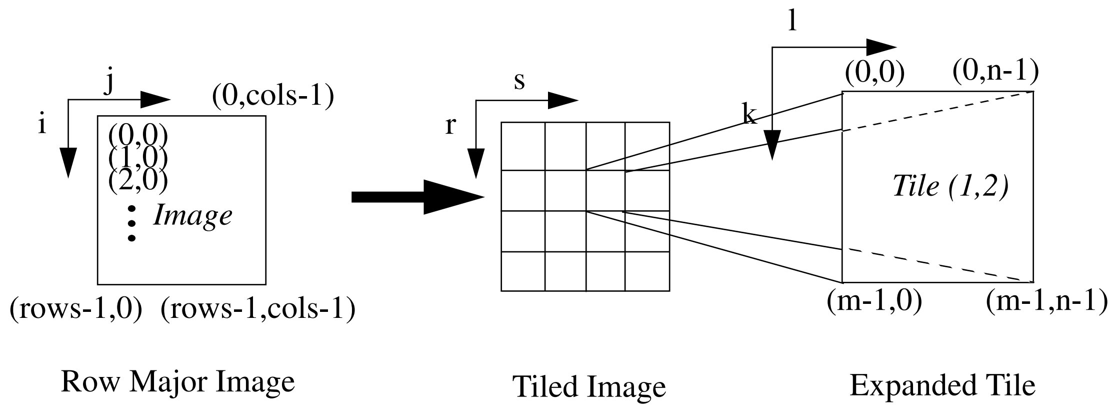

FIGURE 2 Mosaic, Image and Tile Notation

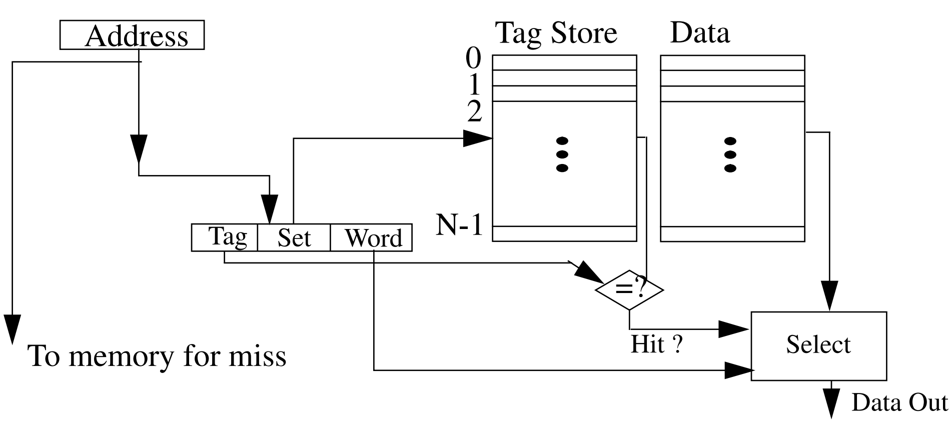

FIGURE 3 Cache Architecture

```
for i = 0 to rows+m-2
   for j = 0 to cols+n-2
          Dilation(i, j)  =  0
          for k = 0 to m-1
             for l= 0 to n -1
                    ii = i-k+k0
                    jj = j-l+l0
                    if (ii < rows && ii>=0 ) && (jj < cols && jj>=0)
                           Dilation(i, j)  =  Max [Dilation(i, j), Image(ii, jj) + SE(k, l)]
```

FIGURE 4 Dilation program

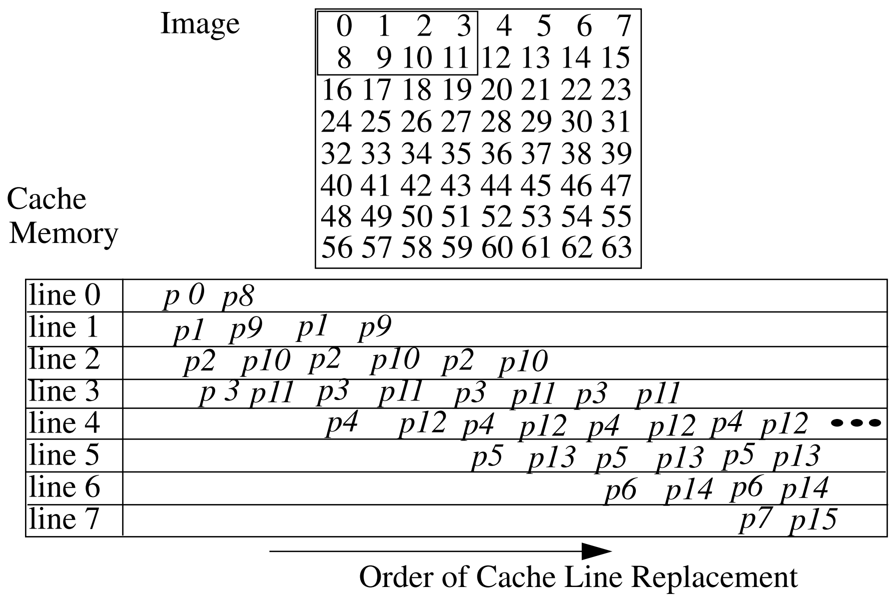

FIGURE 5 Caching into an 8 line direct mapped cache

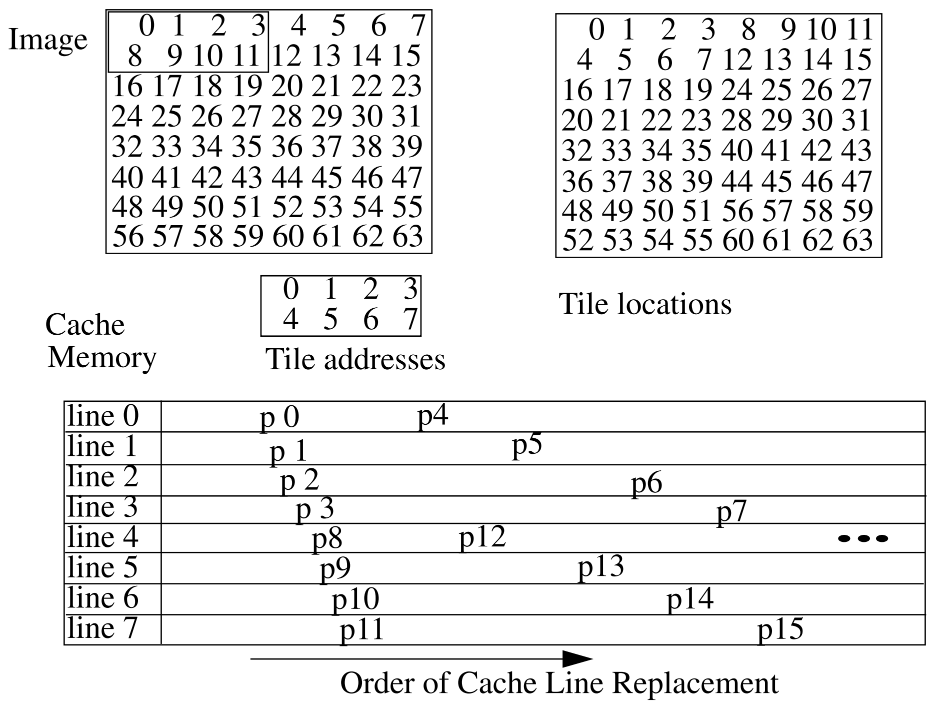

FIGURE 6 Caching into an 8 line cache and lining

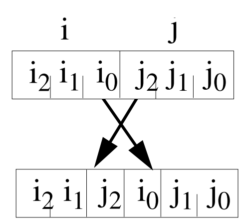

FIGURE 7 Address Transformation For 2x4 tile

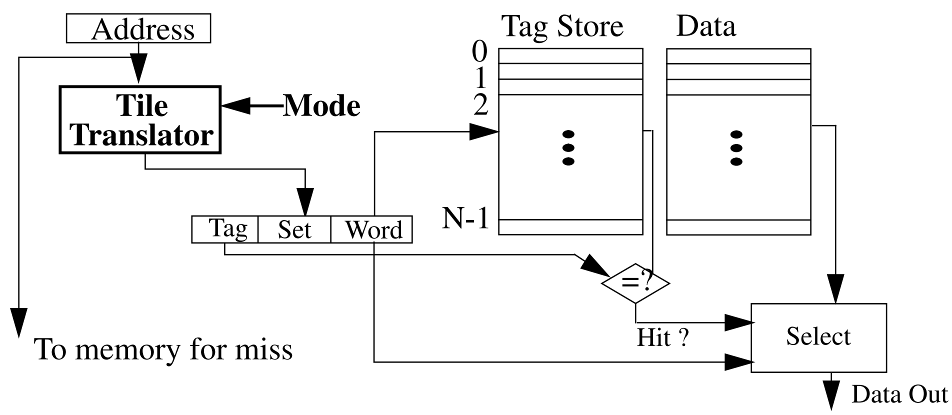

FIGURE 8 The Cache Architecture with Tiling

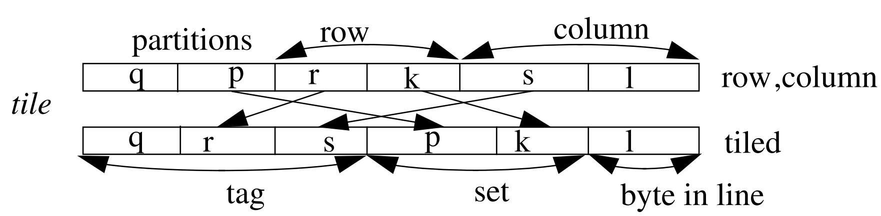

FIGURE 9 Partitioning and Tiling

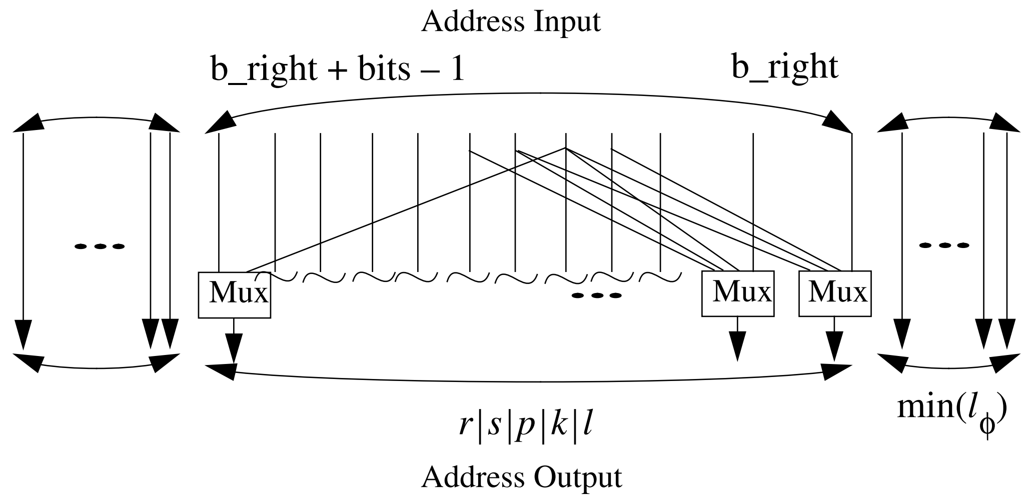

FIGURE 10 Multiplexing address solution

FIGURE 11 Hit Ratios for Tiling and Regular Caches. 256x256 image, Image Reads Only

FIGURE 12 Hit Ratios for Tiling and Regular Caches All Images, Reads, Writes and Structuring Element Reads

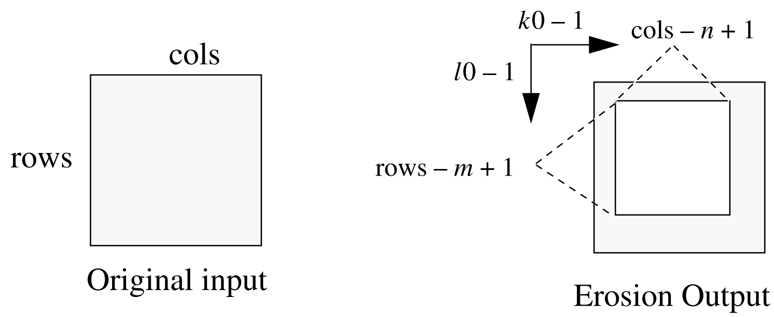

FIGURE 13 Image input and output for erosion

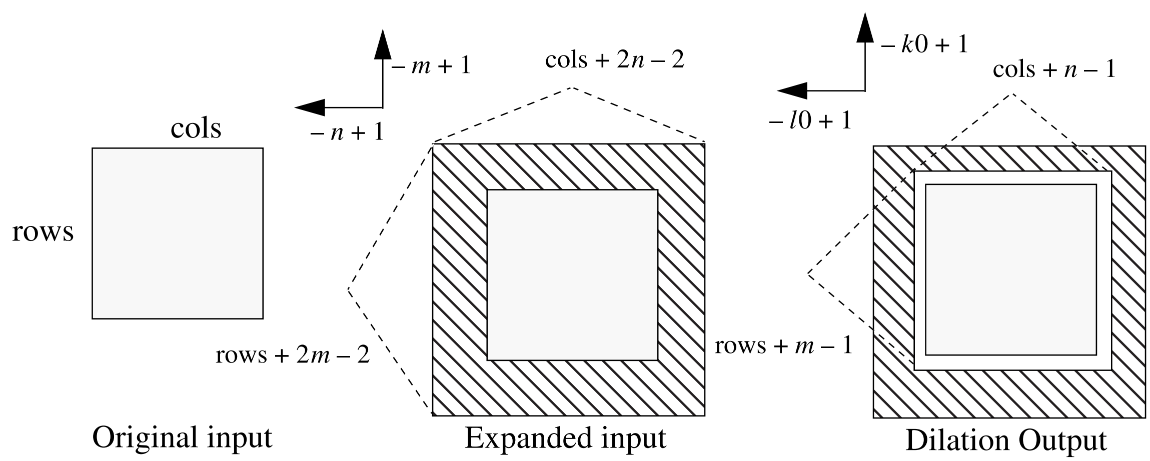

FIGURE 14 Image expansion for input to dilation for $(\text{rows} + m - 1) \times (\text{cols} + n - 1)$ result

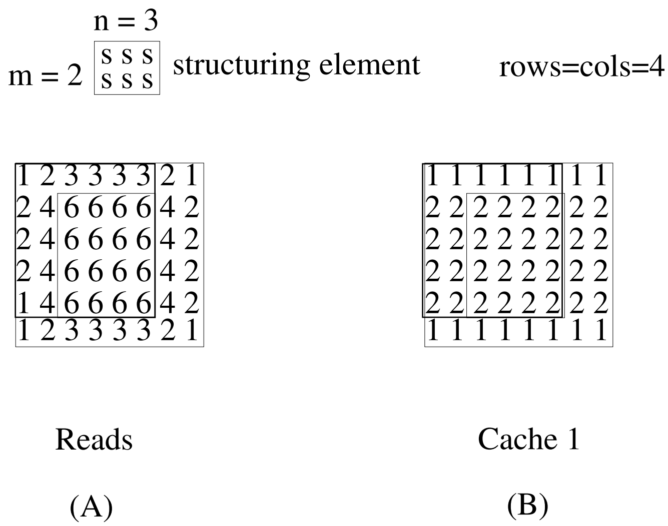

FIGURE 15 Example of cache read misses for full dilation

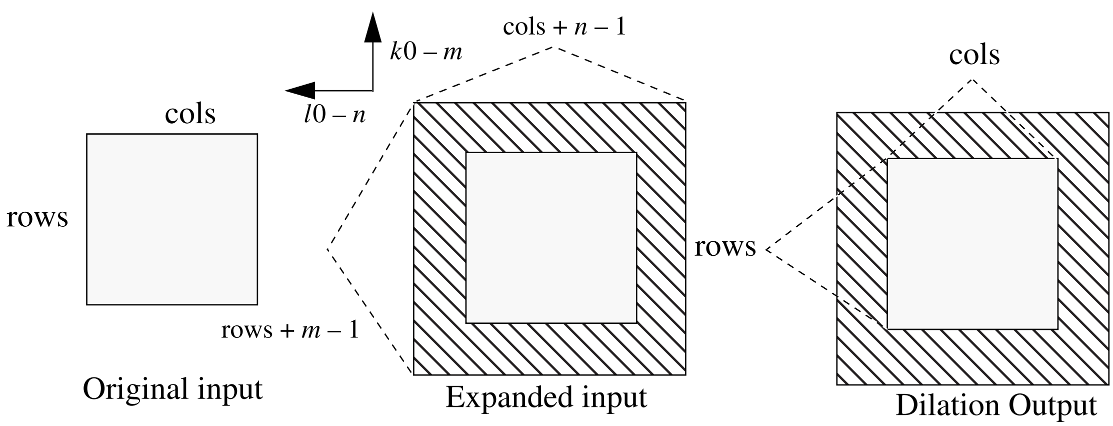

FIGURE 16 Image expansion for input to dilation for $rows \cdot cols$ result

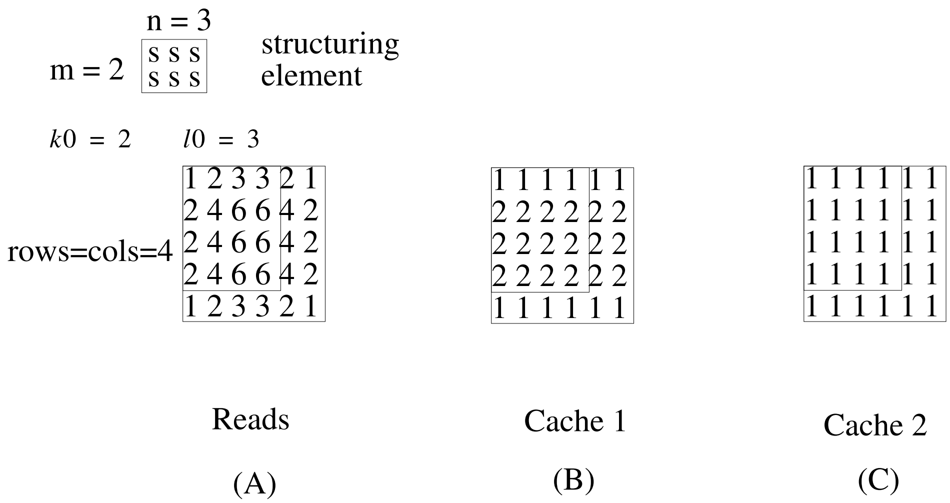

FIGURE 17 Example of Cache Read Misses For Restricted Input and Output

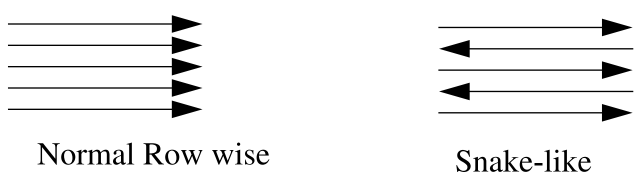

FIGURE 18 Normal Traversal vs. Snake-like