J Sign Process Syst (2011) 62:65–76
DOI 10.1007/s11265-008-0298-z

Real-Time Adaptive Background Modeling for Multicore Embedded Systems

Senyo Apewokin · Brian Valentine · Jee Choi · Linda Wills · Scott Wills

Received: 13 April 2008 / Revised: 12 September 2008 / Accepted: 9 October 2008 / Published online: 5 November 2008
© 2008 Springer Science + Business Media, LLC. Manufactured in the United States

Abstract Current trends in microprocessor design integrate several autonomous processing cores onto the same die. These multicore architectures are particularly well-suited for computer vision applications, where it is typical to perform the same set of operations repeatedly over large datasets. These memory- and computation-intensive applications can reap tremendous performance and accuracy benefits from concurrent execution on multi-core processors. However, cost-sensitive embedded platforms place real-time performance and efficiency demands on techniques to accomplish this task. Furthermore, parallelization and partitioning techniques that allow the application to fully leverage the processing capabilities of each computing core are required for multi-core embedded vision systems. In this paper, we evaluate background modeling techniques on a multicore embedded platform, since this process dominates the execution and storage costs of common video analysis workloads. We introduce a new adaptive backgrounding technique, multimodal mean, which balances accuracy, performance, and efficiency to meet embedded system requirements. Our evaluation compares several pixel-level background modeling techniques in terms of their computation and storage requirements, and functional accuracy for three representative video sequences, across a range of processing and parallelization configurations. We show that the multimodal mean algorithm delivers comparable accuracy of the best alternative (Mixture of Gaussians) with a 3.4× improvement in execution time and a 50% reduction in required storage for optimal block processing on each core. In our analysis of several processing and parallelization configurations, we show how this algorithm can be optimized for embedded multicore performance, resulting in a 25% performance improvement over the baseline processing method.

S. Apewokin (✉) · B. Valentine · J. Choi · L. Wills · S. Wills
Georgia Institute of Technology,
Atlanta, GA, USA
e-mail: senyo@ece.gatech.edu

Keywords Background modeling · Embedded computer vision · Multicore

1 Introduction

Background/foreground segmentation is an important but costly component of many computer vision applications, particularly in video surveillance and analysis. Several techniques that utilize robust background modeling algorithms to identify salient foreground objects have been developed. Typically, the current video frame is compared against a background model representing elements of the scene that are stationary or changing in uninteresting ways (such as rippling water or swaying branches). Foreground is determined by locating significant differences between the current frame and the background model.

The availability of low-cost, portable imagers and new embedded computing platforms makes video surveillance and analysis possible in new environments. However, situations in which a portable, embedded video surveillance system is most useful (e.g., monitoring outdoor and/or busy scenes) also pose the greatest challenges. Real-world scenes are characterized by changing illumination and shadows, multimodal features (such as rippling waves and rustling leaves), and frequent, multilevel occlusions. To extract foreground in these dynamic visual environments, adaptive multimodal background models are frequently used that maintain historical scene information to improve accuracy. These methods are problematic in real-time embedded

environments where limited computation and storage restrict the amount of historical data that can be processed and stored.

Embedded multicore processors present an ideal platform for real-time performance of computer vision applications. These workloads require high bandwidth and computational capacity, therefore executing tasks concurrently on multiple computing cores can yield significant speedup. Due to embedded constraints, however, available memory storage is limited on each embedded computing core. As a result, the workload must be partitioned into blocks for execution and several of these iterations may be necessary to complete the processing of just one frame. Simply dividing existing algorithms into parallel threads to execute on each core may not yield optimum performance because of the overhead of transferring multiple blocks.

In this paper, we examine several representative pixel-based background modeling techniques across a range of processing and partitioning arrangements of a multicore embedded platform. We focus on background modeling algorithms because background/foreground segmentation and background model maintenance typically account for the majority of execution and storage costs of video processing workloads [7]. The backgrounding techniques are evaluated in terms of computational cost, storage, and extracted foreground accuracy. The techniques range from simple, computationally inexpensive methods, such as frame differencing and mean/median temporal filters [8], to more complex methods, including the multimodal Mixture of Gaussians (MoG) [14, 15] approach. In this comparative evaluation, we introduce a new approach, multimodal mean (MM), for real-time background modeling [21]. We show that this technique achieves accuracy comparable to multimodal MoG techniques but with a 3.4× improvement in execution time and a 50% reduction in required storage for optimal block processing on each core.

In our comparative evaluation, we explore how different processing and partitioning arrangements affect each technique on a multicore processor. We first measure the memory storage costs and performance (execution time) for each algorithm using a simple partitioning scheme. With this approach the current frame is partitioned into equal data workloads for processing in the computing cores. In cases where the entire frame cannot be processed in one pass, several iterations may be required to process the entire image. We then explore a tiled partitioning arrangement in which consecutive images are processed in stacks of smaller image blocks on each core. The advantage of this approach is that it minimizes data transfers of the shared background model between main memory and each processing core. We evaluate different configurations of this tiled arrangement and their impact on processing time and storage costs for each algorithm.

Our results demonstrate that our proposed MM algorithm achieves competitive real-time foreground accuracy under a variety of outdoor and indoor conditions on a multicore embedded platform under the tiled processing approach. This configuration achieves a 25% increase in MM algorithm performance over the single buffered baseline approach.

For our testbed, we employ the Cell Broadband Engine (B.E.) on a Sony Playstation 3 running Yellow Dog Linux 5.0. The Cell B.E. is a heterogeneous multicore chip made of a main processor and eight co-processors connected with a high-bandwidth bus. The co-processors are designed for high-performance data-streaming and data-intensive computation and the entire system is well-suited for embedded image processing applications.

The rest of this paper is organized as follows:

In Section 2, we present the motivation for improving the background modeling component of computer vision applications to improve real-time performance. We also discuss the enormous potential for real-time background modeling on multicore systems and present a review of current background modeling techniques. In Section 3, we introduce a new algorithm, called *multimodal mean*, and evaluate it against existing techniques on a multicore system. In Section 4, we show different partitioning techniques to optimize the performance of multimodal mean on a multicore platform. Section 5 concludes the paper.

2 Motivation and Related Work

Current and emerging microprocessor designs integrate several autonomous processing cores onto the same die [1]. Industry efforts, such as the Cell Broadband Engine from Sony, Toshiba, and IBM,[2] Niagara from Sun [3], and Montecito from Intel [4], as well as university-led designs, such as MIT's RAW [5] and the University of Texas's Trips [6] are representative multicore architectures.

Multicore architectures provide tremendous potential to achieve real-time performance of computer vision applications, where the same sets of operations are typically applied repeatedly over large datasets. However, on embedded multicore systems, power, size and other constraints limit the availability of hardware resources. Optimizing algorithms to achieve real-time performance on such systems, while observing embedded constraints, becomes a challenging but necessary task.

2.1 Computer Vision Workload Analysis

Chen et al. [7] present a comprehensive analysis of computer vision workloads. They chose video surveillance as a representative case study of a complex computer vision application and profile it with the Intel VTune Performance Analyzer. Their results show that foreground/background

segmentation is the most expensive module in the workload and accounts for up to 95% of the execution time. According to their analysis, their background modeling algorithm consumes 1 billion micro-instructions for a frame size of 720×576 pixels and takes 0.4 s to execute on a 3.2 GHz Intel Pentium 4 processor. Further analysis of the module shows that about 60% of the background/foreground segmentation time is used for updating and maintaining the background model. This shows that the choices made for pixel representations and the associated learning/adaptation techniques greatly influence both performance and storage costs of the model.

Since a critical component of computer vision applications is background modeling, speeding up this component will greatly improve the real-time performance capabilities of the overall system in accordance with Amdahl's law. We approach this task from two directions:

1. optimizing background modeling algorithms for embedded systems, and
2. optimizing processing and partitioning of background modeling data for multicore systems.

2.2 Related Background Modeling Work

A variety of techniques exists for background modeling; see [8–10] for recent comprehensive surveys. *Frame differencing* compares pixels in the current video frame with corresponding pixels in the previous frame. If the difference between the pixels is above a given threshold, then that pixel is identified as foreground. While computationally inexpensive, this method is prone to the foreground aperture problem [10] and cannot handle dynamic background elements, such as swaying tree branches.

Sliding window-based (or *nonrecursive* [8]) techniques keep a record of the w most recent image frames. The background is represented as the mean or median of the frames in the buffer. Foreground is determined either by determining if the current image pixel deviates by a fixed threshold away from the background model or, if it is within some standard deviation of the background. Although less sensitive to the aperture problem, this type of technique is more memory intensive as it requires w image frames of storage per processed image.

Recursive techniques [8] utilize only the current frame and parametric information accumulated from previous frames to separate background and foreground objects. These techniques typically employ weighted means or approximated medians and require significantly less memory than the sliding window techniques. An *approximated median* is computed in [11]. The background is initialized by declaring the first image frame as the median. When a new video frame is acquired, the current image pixel values are compared with those of the approximated median pixel values. If a pixel value is above the corresponding median value, then that approximate median pixel value is incremented by one, otherwise it is decremented by one. It is assumed that the approximated median frame will eventually converge to the actual median after a given number of image frames are analyzed [11]. In [12] and [13] a *weighted mean* is used, which takes a percentage of the background pixel and a percentage of the current pixel to update the background model. This percentage is governed by a user-defined learning rate that affects how quickly objects are assimilated into the background model.

Issues can arise with the described techniques when there are moving background objects, rapidly changing lighting conditions, and gradual lighting changes. The Mixture of Gaussians (MoG) [14, 15] and Wallflower [10] approaches are designed to better handle these situations by storing *multimodal representations* of backgrounds that contain dynamic scene elements, such as trees swaying in the wind or rippling waves. The MoG approach maintains multiple data values for each pixel coordinate. Each data value is modeled as a Gaussian probability density function (pdf) with an associated weight indicating how much background information it contains. With each new image frame, the current image pixel is compared against the pixel values for that location. A match is determined based on whether or not the current pixel falls within 2.5 standard deviations of any of the pixel distributions in the background model.

Wallflower [10] uses a three-tiered approach to model foreground and background. Pixel, region, and frame-level information are obtained and analyzed. At the pixel-level, a linear predictor is used to establish a baseline background model. At the region-level, frame differencing, connected component analysis and histogram backprojection are used to create foreground regions. Multiple background models are stored at the frame-level to handle a sharp environmental change such as a light being switched on or off.

These techniques have limitations in either foreground extraction accuracy or real-time performance when applied to busy or outdoor scenes in resource-constrained embedded computing systems. Frame differencing and recursive backgrounding methods do not handle dynamic backgrounds well. Sliding window methods require significant memory resources for accurate backgrounding. The MoG approach requires significant computational resources for sorting and computations of standard deviations, weights, and pdfs.

3 Multimodal Mean Background Modeling Technique

In this paper, we present a new backgrounding technique, called *multimodal mean* (MM), that has the multimodal

modeling capabilities of MoG but at significantly reduced storage and computational cost. A related approach by Appiah and Hunter [16] implements multimodal backgrounding on a single-chip FPGA using a collection of temporal lowpass filters instead of Gaussian pdfs. A similar background weight, match, and updating scheme as the MoG is maintained, with simplifications to limit the amount of floating-point calculations. In contrast to these approaches, we use a linear parameter updating scheme as opposed to nonlinear updates of weights and pixel values, and we make use of information about recency of background pixel matches. Updating the background model information in this manner allows for efficient storage of a pixel's long-term history.

3.1 Algorithm

We propose a new adaptive background modeling technique, called *multimodal mean*, which models each background pixel as a set of average possible pixel values. In background subtraction, each pixel I_t in the current frame is compared to each of the background pixel means to determine whether it is within a predefined threshold of one of them. Each pixel value is represented as a three-component color representation, such as an RGB or HSI vector. In the following, $I_{t.x}$ represents the x color component of a pixel in frame t (e.g., $I_{t.\mathrm{red}}$ denotes the red component of I_t). The background model for a given pixel is a set of K mean pixel representations, called *cells*. Each cell contains three mean color component values. An image pixel I_t is a background pixel if each of its color components $I_{t.x}$ is within a predefined threshold for that color component E_x of one the background means.

In our embedded implementation, we chose K=4 cells and use an RGB color representation. Each background cell B_i is represented as three running sums for each color component $S_{i,t.x}$ and a count $C_{i,t}$ of how many times a matching pixel value has been observed in t frames. At any given frame t, the mean color component value is then computed as $\mu_{i,t.x}=S_{i,t.x}/C_{i,t}$.

More precisely, I_t is a background pixel if a cell B_i can be found whose mean for each color component x matches within E_x the corresponding color component of I_t:

$$\left(\bigwedge_x \left|I_{t.x}-\mu_{i,t-1.x}\right| \le E_x\right) \wedge \left(C_{i,t-1} > T_{\mathrm{FG}}\right),$$

where T_{FG} is a small threshold number of times a pixel value can be seen and still considered to be foreground. (In our experiments, T_{FG}=3 and E_x=30, for $x \in \{R,G,B\}$.)

When a pixel I_t matches a cell B_i, the background model is updated by adding each color component to the corresponding running sum $S_{i,t.x}$ and incrementing the count $C_{i,t}$. As the background gradually changes, for example, due to lighting variations), the running averages will adapt as well. In addition, to enable long-term adaptation of the background model, all cells are periodically *decimated* by halving both the sum and the count every d (the decimation rate) frames. To be precise, when I_t matches a cell B_i, the cell is updated as follows:

$$S_{i,t.x} = \left(S_{i,t-1.x} + I_{t.x}\right)/2^b$$

$$C_{i,t} = \left(C_{i,t-1} + 1\right)/2^b ,$$

where b=1 if t mod d=0, and b=0, otherwise.

Decimation is used to decay long-lived background components so that they do not permanently dominate the model, allowing the background model to adapt to the appearance of newer stationary objects or newly revealed parts of the background. It also plays a secondary role in the embedded implementation in preventing counts from overflowing their limited storage.

When a pixel I_t does not match cells at that pixel position, it is declared foreground. In addition, a new background cell is created to allow new scene elements to be incorporated into the background. If there are already K background cells, a cell is selected to be replaced based on the cell's overall count $C_{i,t}$ and a recency count $R_{i,t}$ which measures how often the background cell's mean matched a pixel in a recent window of frames. A sliding window is approximated by maintaining a pair of counts ($r_{i,t}$, $s_{i,t}$) in each cell B_i. The first $r_{i,t}$, starts at 0, is incremented whenever B_i is matched, and is reset every w frames. The second $s_{i,t}$, simply holds the maximum value of $r_{i,t}$ computed in the previous window:

$$r_{i,t} = \begin{cases} 0, & \text{when } t \mod w = 0 \\ r_{i,t-1}+1 & \text{when } B_i \text{ matches } I_t \text{ and } t \,\mathrm{mod} w \neq 0 \end{cases}$$

$$s_{i,t} = \begin{cases} r_{i,t-1}, & \text{when } t \bmod w = 0 \\ s_{i,t-1}, & \text{otherwise.} \end{cases}$$

Recency $R_{i,t} = r_{i,t} + s_{i,t}$ provides a measure of how often a pixel matching cell B_i was observed within a recent window. The $s_{i,t}$ component allows information to be carried over across windows so that recency information is not completely lost at window transitions. When a new cell is created and added to a background model that already has K cells, the cell to be replaced is selected from the subset of cells seen least recently, i.e., cells whose recency $R_{i,t}$ < w/K. From this set, the cell with the minimum overall count $C_{i,t}$ is selected for replacement. If all cells have a recency count $R_{i,t}$>w/K (in the rare event that all cells are observed equally often over an entire

window), then the cell with lowest $C_{i,t}$ is replaced. (In our experiments, we chose w=32.)

3.2 Processing and Storage Costs

Pixel-level image processing algorithms typically involve a small number of micro-operations performed over a large number of pixels. This makes them memory-intensive as well as compute-intensive. For example, a 720×640 pixel image in a standard RGB format requires 1.38 MB to store the raw image for further processing. Applying a single unary operation to each pixel in the image contributes 460,800 operations to the entire execution.

Background modeling algorithms, which are a subset of pixel-level image processing algorithms, are characterized by high memory requirements, large numbers of micro-operations and little data reuse. Memory is required to store the current frame being processed as well as the background model which typically includes representations for each pixel in the image. For the same image in the example above, adding a single byte field to a given pixel representation in the background model increases the size by 460 KB or a third the input image size.

For most systems, it will be nearly impossible to perform the entire background/foreground segmentation of a typical image without repeated block transfers of image data. A multicore system allows the processing of different parts of the image to proceed concurrently. On embedded systems, however, limited memory decreases the processing block size and therefore more iterations are required.

Data domain parallelization [17], where the data is partitioned into independent pieces which are processed by each core executing the entire algorithm, is most suitable for background modeling workloads. This is more preferable than dividing the algorithm into separate functions (*function domain parallelization* [17]) because the relatively few number of operations performed per-pixel does not offset the modularization overhead. Also, there will be extra transfer overhead encountered when moving the partially updated background models between cores for each processing step.

It is also noteworthy that the memory access patterns for this workload are very predictable. It is therefore more desirable to handle memory transfers to execution cores directly through the application program rather than through more generalized underlying hardware such as caches [18].

3.3 Experiment

We evaluate a set of background modeling techniques using two representative test sequences executing on an embedded multicore platform. Each technique is compared in terms of image quality and accuracy (false positives and false negatives) as well as execution cost (execution time and storage required). The evaluated techniques include:

- frame differencing
- approximated median
- weighted mean
- sliding window median
- sliding window mean
- mixture of Gaussians (MoG)
- multimodal mean (MM)

The test suite is comprised of two long outdoor sequences captured using an inexpensive webcam (see Table 1). All sequences have a frame size of 720×640. The test sequences were chosen because they contain scenarios that present difficult challenges for background modeling algorithms.

The “Outdoors I” scene involves a busy pedestrian walkway outlined by trees and was taken on a sunny day. Under those real-world conditions the background model must deal with distracting features and uninteresting motion resulting from waving trees and shadows.

The second outdoor scene “Outdoors II” was chosen for its fluctuating illumination condition which is another key challenge for background modeling algorithms running in real-world environments. This video also contains the continuous presence of foreground objects in the periphery of the image which could result in a noisy segmentation. Both videos were recorded at 30 frames per second (fps) and down sampled to 1 fps for processing.

Table 2 lists the algorithm parameters used in the experiments. Experiment parameters and thresholds were held constant for all sequences. The MoG method incorporated K=4 Gaussians while the MM method utilized K=4 cells. The sliding window implementations use a buffer size of four for comparable memory requirements.

Our execution platform is a Sony Playstation 3 with the Cell B.E. multicore processor running Yellow Dog Linux 5.0. The Cell B.E. is a heterogeneous multicore processor made up of two distinct types of processing cores: the power processing unit (PPU) and the synergistic processing unit (SPU). In the Playstation 3, one PPU and six SPUs are available for application development.

The PPU is the main processor and is a fully compliant 64-bit PowerPC general-purpose processor with 32 128-bit vector registers, 32-KB L1 instruction and data caches, and

Table 1 Test sequences.

Sequence	# frames	Sampled frame
Outdoors I	700	453
Outdoors II	700	453

Table 2 Algorithm parameters.

Algorithm	Parameters
Mean/median (SW)	|Window|=4
Weighted mean	α=0.1 for u_t=$(1-\alpha)\times u_{t-1}+\alpha x_t$
Mixture of Gaussians (MoG)	K=4 modes, initial weight w=0.02, learning rate α=0.01, weight threshold T=0.85.
Multimodal mean	K=4, E_x=30 for $x\in\{R, G, B\}$, T_{FG}=3, d=400, w=32

Table 3 Memory allocation.

Algorithm	Block size (pixels)	Image size (KB)	BG model size (KB)
Frame differencing	38,400	115.2	115.2
Approximated median	38,400	115.2	115.2
Weighted mean	12,800	38.4	153.6
Median (SW)	38,400	115.2	115.2
Mean (SW)	12,800	38.4	153.6
MoG	1,600	4.8	160
Multimodal mean	3,200	9.6	153.6

a 512-KB unified L2 cache. In our experiments, we use this processor for image decoding and encoding, core synchronization and other book-keeping tasks.

We use the SPUs which are designed for high-performance data-streaming and data-intensive computation to perform the bulk of the background modeling algorithms. Each SPU is a 128-bit RISC processor with 128 128-bit registers and 256 KB of local storage. The SPUs are not cached-based and DMA is the primary method of communication between the SPUs and main memory. The maximum size of each DMA transfer is 16 KB. The Element Interconnect Bus (EIB), which is a very high-speed, high bandwidth communication network, provides a critical communication link between the powerful computing cores and main memory.

Although we perform our evaluation on a single platform the results can be generalized across other multicore embedded platforms. On a homogenous multicore chip, one of the cores will be dedicated to obtaining the images either through a driver attached to a camera or by decoding images retrieved from main memory. We believe most processing cores should capably handle this dedicated task so there is no added benefit of having the PPU on the Cell. Also, the SPUs which are responsible for much of the core processing have only 256 KB of local storage. Limited on-chip memory on the image processing cores is representative of a true embedded system. For systems with smaller memories the advantages of the reduced storage features of our algorithm and the benefits of our processing and partitioning techniques will be more pronounced.

We implemented the background modeling algorithms in C and compiled them using gcc for the PPU and gcc-spu for the SPU. The background model is created and maintained by the PPU and different parts are transferred to each SPU along with the corresponding image to process. This arrangement is necessary because even the least memory intensive background modeling techniques (e.g. frame differencing) could not support the entire 720× 640 image being processed by all the computing cores in one pass. For the multimodal mean algorithm, the periodic decimation and recency resets are performed by the PPU and all other components of the algorithm are performed on the SPU. For all the other techniques, the entire algorithm is run on the SPU.

All images from the test sequences are in JPEG format and these are loaded onto the hard drive before each run. We use the independent JPEG library [19] to perform the image encoding and decoding.

3.4 Evaluation and Results

In this discussion, block size is the portion of a given image that is processed by a single SPU in a single iteration. In this definition iteration represents one complete cycle of image loading, processing, and write back. To keep the processing balanced among cores we limit the chosen block sizes using the constraint

$$\text{Image Size mod (Block Size} \times \text{Number of SPU)} = 0,$$

where Image and Block Sizes are measured in pixels.

For our first evaluation, we chose the configuration that maximizes block size. We divide the SPU storage into two

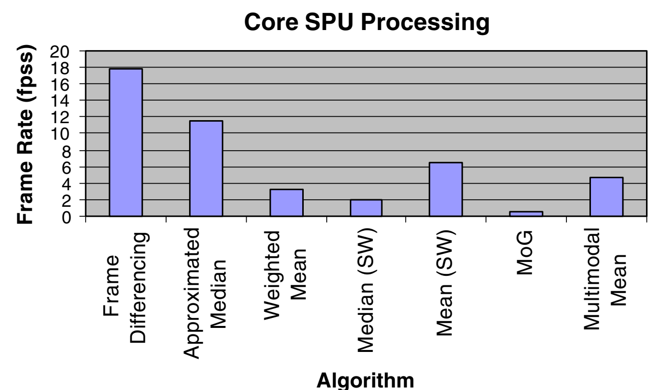


Figure 1 Algorithm performance in frames per spu seconds, excluding data transfer latency.

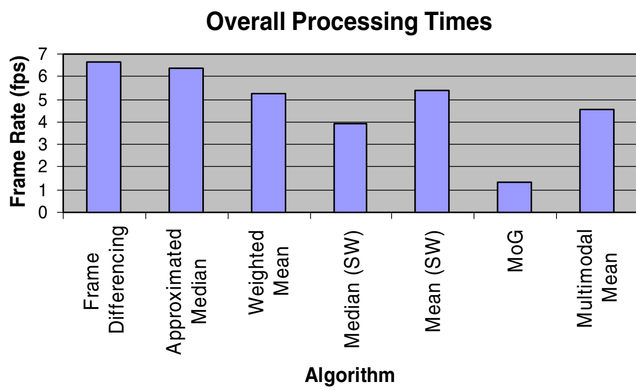


Figure 2 Algorithm performance in frames per second, including data transfer latency.

areas; the first holds a block of the current frame and the other holds the corresponding portion of the background model. It is noteworthy that these storage areas are of equal size bytewise for the single mode background models but vary for the other multi-modal models.

Table 3 shows the memory allocation on each SPU using this configuration. The multimodal background models require storage for four modes per pixel making them significantly larger than the single mode models. This minimizes the block size for those techniques. Also, the MM technique uses only integer storage types as opposed to MoG which uses floating-point and has half the block size.

Figures 1 and 2 show the performance results obtained from running each algorithm on the configuration described in Table 3. Because the sequence of frames originates from standard files rather than camera output, I/O requirements are not included in these figures.

Figure 1 shows the core algorithm performance results excluding data transfer latency for each algorithm. This includes processing times from the time the data is available on each core and the algorithm begins to the time the algorithm is completed. Using the spu_decrementer [20], we recorded the time spent executing the core algorithm on each SPU. Results are given in frames per

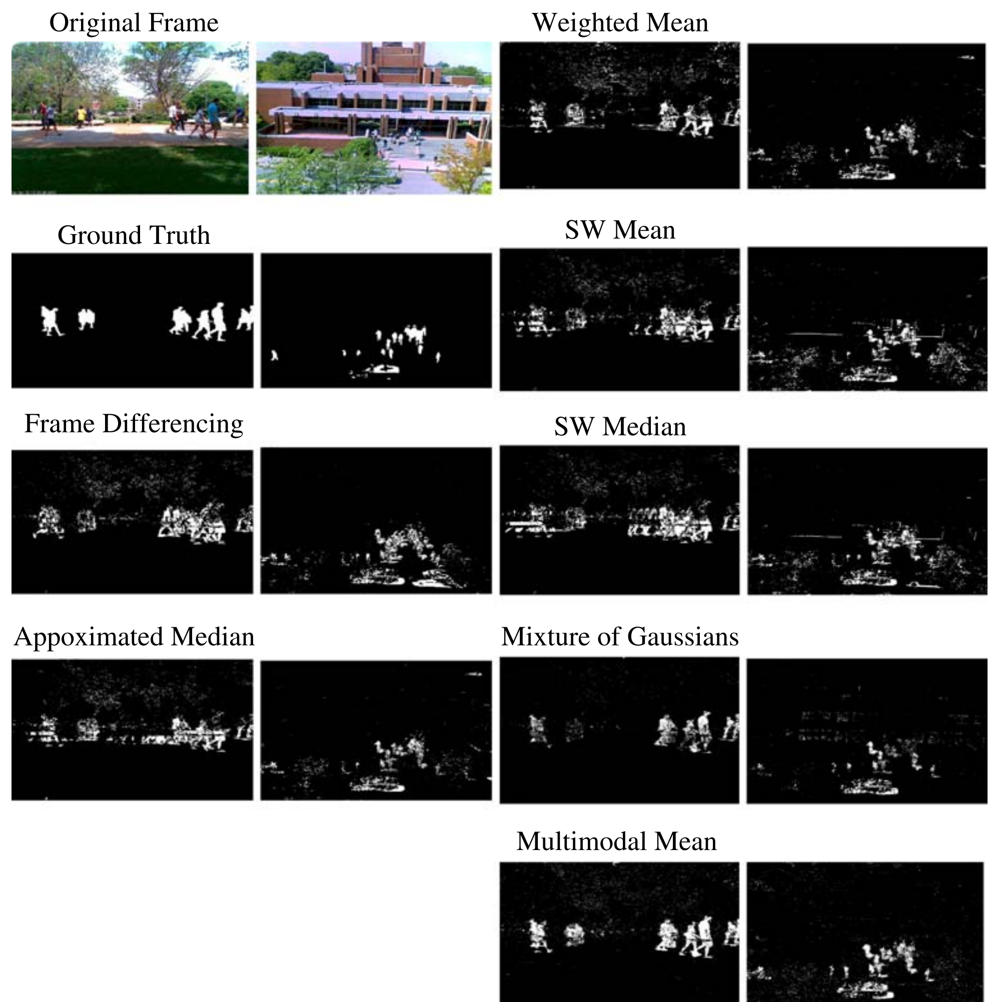


Figure 3 Results of extracting foreground using different backgrounding techniques.

spu seconds (fpss). The frame rates displayed are those for the slowest SPU among the cores although there is not significant variation in SPU times. From the results shown in Figure 1, we observe that the techniques with fewer operations, such as frame differencing and approximated median, generally run faster than the multimodal ones. MM has comparable performance to other sliding window techniques and has about nine times better performance than MoG. This is due to MoG's increased complexity and more costly floating-point computations. These results are consistent with results obtained on an eBox 2300 Vesa PC, which is an uniprocessor embedded platform [21].

Figure 2, shows the overall algorithm performance in frames per second, including data transfer. Overall, our results show that MM achieves a 3.4× speedup over MoG and has comparable performance to the other techniques. In general, we observe that the disparity between the performance of single mode techniques and that of the multimodal ones, particularly MM, is narrowed. There are two reasons. First, there is a comparatively higher data transfer latency associated with the single mode techniques. Table 2 shows that to process each block these techniques transfer 115.2 KB for the image and another 115.2 KB for the background model. Completing the concurrent transfer of this data to six SPUs in 16 KB chunks results in collisions and all the data must be available on the SPU to begin core processing. Alternatively, the MoG technique for example, transfers only 4.8 KB of image data during each iteration and this is completed in a single DMA transaction.

Second, the ratio of core algorithm execution time to data transfer latency time is higher for the single mode techniques. This results in a disproportionate increase in overall processing times for the single mode techniques as compared to the multimodal ones.

Figure 3 shows the image quality for each backgrounding technique. Multimodal methods (MoG and MM) generally exhibit the lowest number of errors across the sequences. False positives are significantly lower for the multimodal methods.

In "Outdoors I", only the multimodal techniques incorporate the moving trees into the background. Also, the sliding window techniques are less adaptive to the changing foreground and leave a trail behind moving objects. "Outdoors II" features a large number of foreground elements as well as moving trees and MoG and MM handle these scenarios relatively better than the other techniques.

Figure 4a and b quantitatively summarize accuracy for each technique. False positives indicate foreground identified

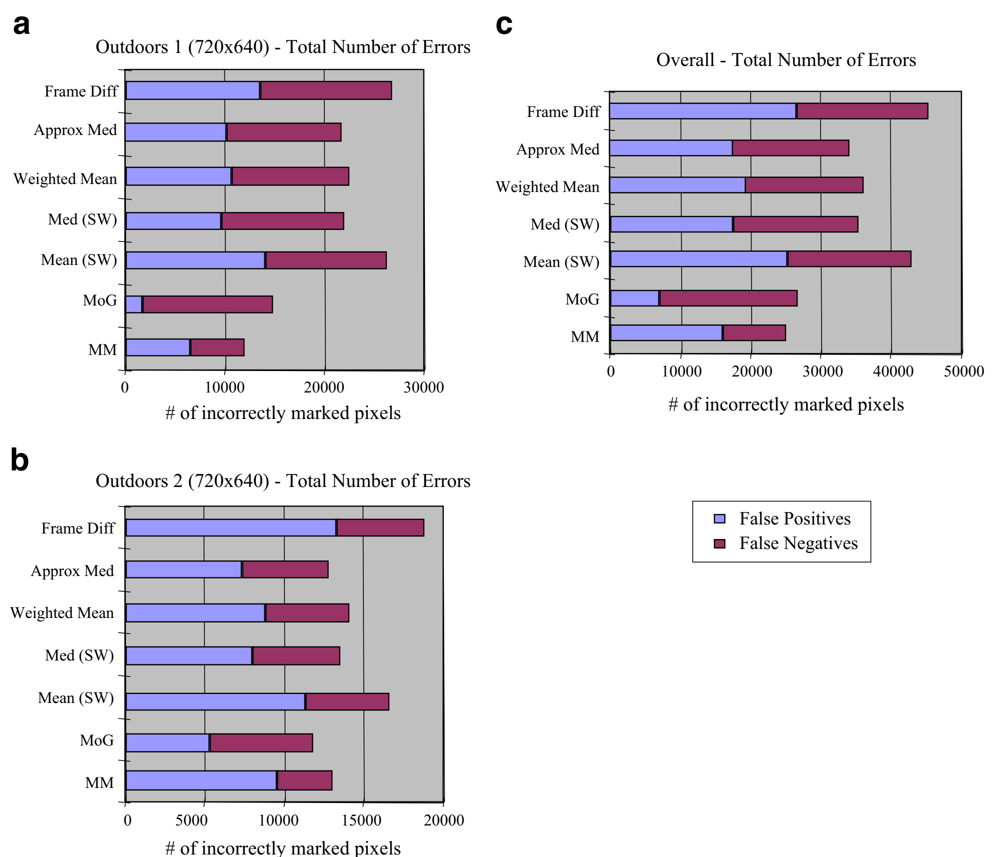


Figure 4 **a** Outdoors I errors. **b** Outdoors II errors. **c** Overall errors.

outside the highlighted (white) regions of the ground truth. False negatives result from background detected in ground truth identified foreground. While these counts do not provide a complete measure of foreground usefulness (e.g., often incomplete foreground can be "filled in"), lower numbers of false positives and negatives are usually desirable. Generally, MoG and MM demonstrate comparable accuracy that is superior to the other methods as is shown in Fig. 4c. In [21], a further evaluation of the accuracy of these techniques on an eBox 2300 Vesa PC embedded platform using additional standard sequences is performed.

4 Tile Processing

For our second evaluation, we examine a different processing arrangement. Typically, live video input from the webcam is buffered as images by the camera driver. Rather than divide the workload in the current frame using maximum block size and available memory on the SPUs, we create a tiled workload from the buffer using much smaller block sizes called *tiles*. Tile sizes are selected analogous to block sizes according to the constraint:

$$\text{Image Size mod } (\text{Tile Size} \times \text{Number of SPU}) = 0,$$

where Image and Tile Sizes are measured in pixels.

We consider a buffer of m of these images. An $n\times k$ tile is selected from the same location for each image in the buffer as shown in Fig. 5a. n is the width of the tile and k is its height. This constitutes the workload for the first image processing core. The next tile location is used for the next core's workload and the process is repeated for all the cores as shown in Fig. 5b.

To tile images as described above, the buffering of the current pixel stream from the imagers has to be modified in the camera driver. Figure 5c shows the stream buffered in receiving order as a continuous array, and depending on the system and availability, this could be a contiguous block of memory. Figure 5d shows the stream broken up into groups of $n\times k$ pixels. The first group of $n\times k$ pixels in the first image is stored at the beginning of the storage array, after which $(m-1)\times n\times k$ pixel locations are skipped in the array before storing the next group. This process is repeated for the entire image stream. The first group of $n\times k$ pixels in image j begins at pixel location $((j-1)\times n\times k)+1$ in the array which was left blank during the buffering of the previous $j-1$ images. This process is repeated for all the images in the buffer.

Tiling the input images as described above could increase the buffering memory latency in the driver so we evaluate the effect of using this technique on our platform.

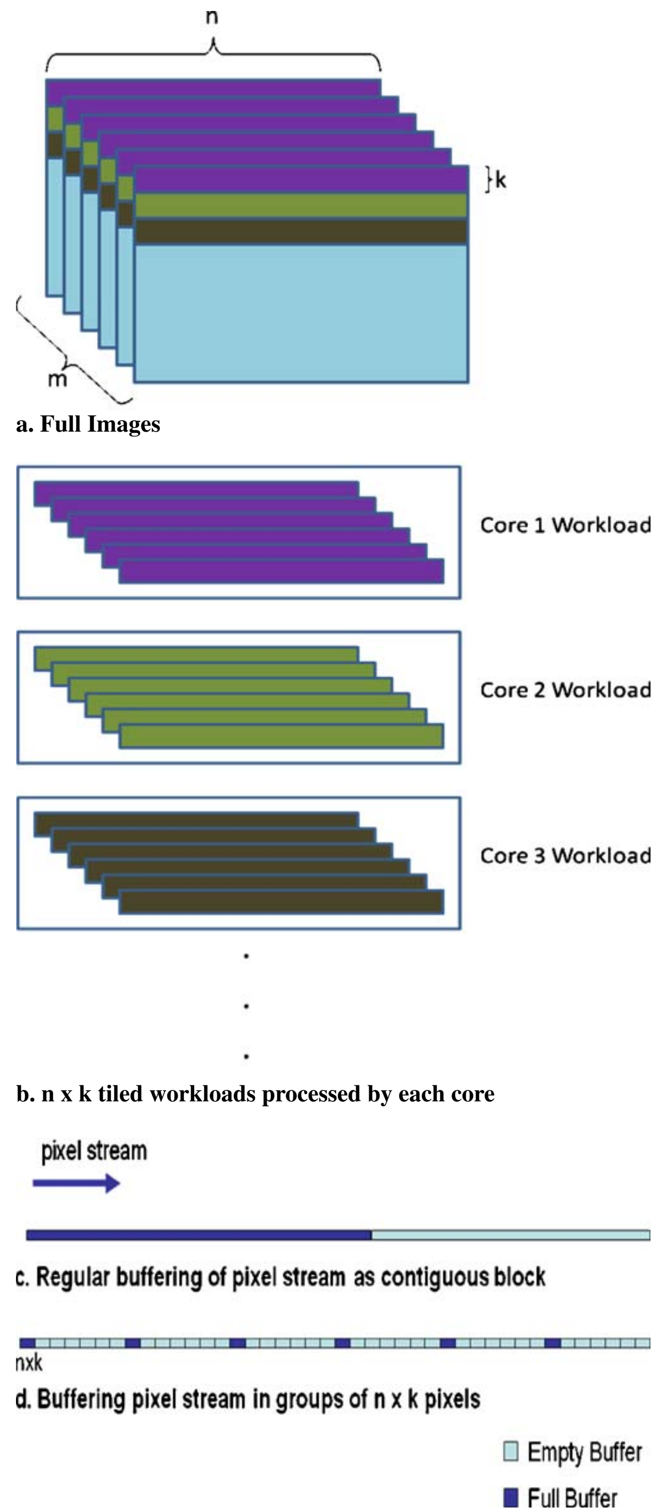


Figure 5 Tiling. **a** Full images. **b** $n\times k$ tiled workloads processed by each core. **c** Regular buffering of pixel stream as contiguous block. **d** Buffering pixel steam in groups of $n\times k$ pixels.

We benchmark the tiled buffering method against the regular buffering method in the jpeg decoder.

The independent jpeg library's decoder generates pixels in rows called scanlines each of which is the width of the

Table 4 Buffering times.

Buffer size	2	16	32
Tile buffering (s)	0.0869	0.0877	0.0881
Stream buffering (s)	0.0896	0.0892	0.0901

image. We stored each scanline using both techniques described above and evaluated the total time taken to decode and buffer each frame. For our evaluation, we used 720×640 images and took the average from decoding 100 images.

Table 4 shows our results for different buffer sizes and suggests that the tile buffering does not significantly increase the image decode times. On average, we record a 0.023% increase in decode time per frame when using tile buffering. We also do not notice an increasing disparity in total time as we increase the buffer size.

Our image retrieval times are significantly less than our background/foreground segmentation processing times for each frame. Slightly increasing the retrieval time does not impact the overall performance, since the two processes are concurrent.

The tile processing configuration minimizes data transfers of the background model between main memory and each SPU by processing a given number of consecutive frames against a single shared background model. Because the tiled workload consists of consecutive frames with the same portion of the image, a single background model is required for processing as well as updating.

We evaluate the multimodal mean technique using this processing configuration for various buffer and tile sizes. We limit our buffer size to 32 which represents reasonable buffering latency on a real-time system.

Table 5 shows the storage requirements for each configuration. All configurations use at most 230 KB of the SPU storage to allow for run time memory requirements.

Table 5 Image storage requirements.

Buffer size	Image size (KB)		
	Tile=1,600 BGM= 76.8 KB	Tile=2,400 BGM= 115.2 KB	Tile=3,200 BGM= 153.6 KB
1	4.8	7.2	9.6
2	9.6	14.4	19.2
4	19.2	28.8	38.4
8	38.4	57.6	76.8
16	76.8	115.2	
32	153.6		

Table 6 Performance.

Buffer Size	Frame rate (fps)		
	Tile=1,600 BGM= 76.8 KB	Tile=2,400 BGM= 115.2 KB	Tile=3,200 BGM= 153.6 KB
1	4.46	4.45	4.46
2	4.48	4.54	4.56
4	4.55	4.66	4.69
8	4.65	4.82	4.95
16	4.8	5.3	
32	5.6		

Table 6 shows the performance for each configuration. The results show a trend of increasing frame rates as the buffer size is increased due to the fewer number of background model transfers to each SPU core. A tile size of 1,600 pixels allows the processing of 32 images in a single transfer and gives the best performance. This configuration achieves a 25% increase in performance over the single buffered baseline approach.

Using the tile buffering technique will cause a multicore video surveillance system to incur a slight buffering latency due to the temporal buffering and, as a consequence, the reaction time of the system could increase slightly. However, the resulting increase in processing bandwidth more than compensates for this one time latency charge which is a small fraction of the overall processing time. Leveraging the multicore resources allows the tiling to be done concurrently with processing. This results in a single temporal buffering latency charge rather than an accumulated latency charge for each frame processed in a uniprocessor system.

More importantly, the bottleneck for computer vision applications on embedded platforms is not the video buffering. The Playstation Eye USB Camera for the Playstation/Cell B.E. platforms buffers full-resolution 640×480 images at 60 fps. Using the tiling technique with buffer size 32 will result in a delayed system reaction time of 0.5 s which is acceptable for a video surveillance system. Depending on the application, the resolution can be decreased or the buffering frame rate increased to achieve even faster reaction time. For example, buffering 320×240 images will result in a delay of only 0.125s.

However, the benefits of the tiling technique are evident with the increased processing bandwidth. For example, an end-to-end application running at 22 fps will be able to run at 30 fps using the tiling technique and a buffer of 32 frames. This is a significant performance increase and yields real-time performance which is crucial to the deployment of real-world embedded vision systems.

5 Conclusion

This paper compares several backgrounding techniques for time sensitive processing on embedded computing platforms. In this context, we evaluate a multimodal mean technique that combines the multimodal features of the mixture of Gaussians with simple pixel evaluation computations involving sums and averages. The multimodal mean method is able to achieve faster execution and lower storage requirements than mixture of Gaussians while providing comparable accuracy and output image quality. We also showed that by partitioning the workload into consecutive tiles of smaller blocks we can further improve the processing times on multicore systems.

Acknowledgements We are grateful for the insightful comments of the anonymous reviewers.

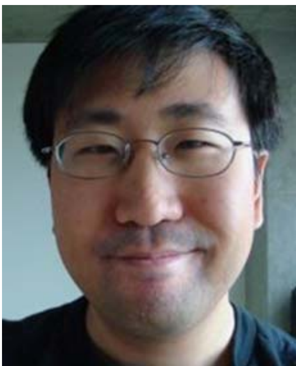
References

1. Bower, F. A., Sorin, D. J., & Cox, L. P. (2008). The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro Magazine*, *28*(3), 17–25. May/Jun.
2. Gschwind, M. (2007). The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, *35*(3), 233–262. June.
3. Kongetira, P., Aingaran, K., & Olukotun, K. (2005). Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro Magazine*, *25* (2), 21–29. March–April.
4. McNairy, C., & Bhatia, R. (2005). Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro*, *25*(2), 10–20. March–April.
5. Taylor, M. B., et al. (2002). The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro Magazine*, *22*(2), 25–35. March–April.
6. Sankaralingam, K., et al. “Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture,” in *Proc. 30th Annual Int. Symp. On Computer Architecture*, pp. 422–433, June 2003.
7. Chen, T. P., Haussecker, H., Bovyrin, A., Belenov, R., Rodyushkin, K., Kuranov, A., Eruhimov, V., “Computer Vision Workload Analysis: Case Study of Video Surveillance Systems”, *Intel Tecnology Journal* 2005.
8. Cheung, S., & Kamath, C. (2004). Robust techniques for background subtraction in urban traffic video. *Video Communications and Image Processing*, *5308*, 881–892. SPIE Electronic Imaging, San Jose, January.
9. Piccardi, M. (2004). Background subtraction techniques: A review. *IEEE International Conference on Systems, Man and Cybernetics*, *4*, 3099–3104. October.
10. Toyama, K., Krumm, J., Brummitt, B., Meyers, B. (). Wallflower: principles and practices of background maintenance,” in *Proc. of ICCV (1),* pp. 255–261, 1999; Wallflower benchmarks available online at research.microsoft.com/~jckrumm/WallFlower/TestImages.htm
11. McFarlane, N., & Schofield, C. (1995). Segmentation and tracking of piglets in images. *Machine Vision and Applications*, *8*(3), 187–193.
12. Jabri, S., Duric, Z., Wechsler, H., & Rosenfeld, A. (2000). Detection and location of people in video images using adaptive fusion of color and edge information. *IEEE International Conference on Pattern Recognition*, *4*, 627–630 September.
13. Wren, C. R., Azarbayejani, A., Darell, T., & Pentland, A. P. (1997). Pfinder:real-time tracking of human body. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *19*(7), 780–785. July.
14. Stauffer, C. & Grimson, W. E. L. (1999). Adaptive background mixture models for real-time tracking”, *Computer Vision and Pattern Recognition*, pp 246–252, June.
15. Stauffer, C., & Grimson, W. E. L. (2000). Learning patterns of activity using real-time tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *22*(8), 747–757. August.
16. Appiah, K., Hunter, A. (2005). A single-chip FPGA implementation of real-time adaptive background model. *IEEE International Conference on Field-Programmable Technology*, pp. 95–102, December.
17. Chen, T., Budnikov, D., Hughes, C., Chen, Y.-K. (2007). Computer vision workloads on multicore processors: articulated body tracking”, *ICME 2007*, Beijing, China, July.
18. Zinner, C. & Kubinger, W. (2006). “ROS-DMA: A DMA double buffering method for embedded image processing with resource optimized slicing,” in *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 361–372, April.
19. T.G. Lane. Using the IJG JPEG Library. Independent JPEG Group, 6b edition, March 1998.
20. Cell Broadband Engine Programming Tutorial http://www-01.ibm.com/chips/techlib/techlib.nsf
21. S. Apewokin, B. Valentine, S. Wills, L. Wills, A. Gentile, “Multimodal Mean Adaptive Backgrounding for Embedded Real-Time Video Surveillance,” *Embedded Computer Vision Workshop (ECVW07),*, June 2007.

Senyo Apewokin is a Ph.D. candidate in the School of Electrical and Computer Engineering at Georgia Institute of Technology. His research interests lie in embedded computer vision, parallel programming architectures, algorithms and frameworks, and multimedia systems. He received his B.S. from Louisiana State University

(2001) and his M.S. from Georgia Tech (2003). In the summer of 2006, he worked as a microprocessor design engineer with Texas Instruments Inc.

Brian Valentine is currently a Ph.D. student in the School of Electrical and Computer Engineering at the Georgia Institute of Technology. He received his B.S. in electrical engineering (2003) and Master of Engineering (2005) degrees at Morgan State University in Baltimore, MD. He has completed multiple internships at Texas Instruments Inc., working on digital signal processor (DSP) programming optimization and application benchmarking. His research interests include algorithms for embedded computer vision, image processing, and embedded code optimization.



Jee W. Choi was born in Seoul, Republic of Korea in 1979. He received his B.S. and M.S. degrees in computer engineering from Georgia Institute of Technology, Atlanta, Georgia, U.S.A in 2000 and 2004 respectively. Currently, he is working towards the Ph.D degree in Electrical and Computer Engineering at Georgia Institute of Technology. His major research interests include embedded video surveillance and numerical computation on multicore architectures.

Linda Wills is an Associate Professor of Electrical and Computer Engineering at the Georgia Institute of Technology, where she was the first recipient of the Demetrius T. Paris, Jr. Professorship. She received her S.B. (1985), S.M. (1986), and Ph.D. (1992) degrees from the Massachusetts Institute of Technology. Dr. Wills served as general chair of the IEEE International Workshop on Rapid System Prototyping (RSP2003) and as program chair of RSP2001. She has also served as general chair and as program chair of the Working Conference on Reverse Engineering. She is a member of the ACM and a senior member of the IEEE and IEEE Computer Society. Her primary research interests are in embedded computer vision and surveillance systems, software understanding and retargeting for embedded systems, parallelizing multimedia applications, and innovative computing systems education.

Scott Wills is a Professor of Electrical and Computer Engineering at Georgia Tech. His research focuses on embedded vision systems, parallel architectures and applications, and supercomputer interconnection networks. He has published 140+ peer-reviewed conference and journal papers in these areas. He earned his B.S. in Physics (1983) from Georgia Tech, and his S.M. (1985), E.E. (1987), and Sc.D. (1990) in Electrical Engineering & Computer Science from MIT. Prior to joining Tech in 1991, Dr. Wills worked at Motorola (Plantation, Florida), Harris (Melborne, Florida), and Symbolics (Cambridge, Massachusetts).