Efficiently Mapping High-Performance Early Vision Algorithms onto Multicore Embedded Platforms

A Dissertation
Presented to
The Academic Faculty

by

Senyo Apewokin

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical Engineering

Georgia Institute of Technology
May, 2009

Efficiently Mapping High-Performance Early Vision Algorithms onto Multicore Embedded Platforms

Approved by:

Date Approved : 16th December 2008

ACKNOWLEDGEMENTS

This dissertation is the culmination of an academic journey spanning several years in which I was deeply influenced by several people. I will be eternally grateful to them for their unyielding support.

First of all, I would like to express my deepest appreciation to Dr. D. Scott Wills, my academic adviser, for his guidance throughout this process. Thank you for allowing me to expend mental energy on tangential albeit interesting ideas while keeping me focused enough on the greater goals.

I would also like to acknowledge the tremendous contributions of my co-adviser Dr. Linda M. Wills. Thank you for teaching me to use the tools of good organization, clarity of thought and expression, and attention to detail for sound research.

I am sincerely grateful to them for guiding the many insightful discussions, topic iterations, platform experimentations, and framing progressions throughout this work. More importantly, I am grateful for the many valuable life-lessons they taught me throughout my association with them.

I am grateful to Dr. Jim Hamblen, Dr. Jeff Davis, Dr Aaron Lanterman, and Dr. David Bader for serving on my thesis committee. Thank you for your numerous platform suggestions and many insightful comments about the general direction of this work. Your advice has certainly resulted in a higher quality document.

I am also grateful to Dr. Hsien-Hsin Lee and Dr. Doug Bough for giving me to alternate research perspectives and Dr. Bonnie Heck and Dr. Sudha Yalamanchili for allowing me to develop educational material.

I am thankful that several current and former PICA and EASL members provided a resourceful work environment during the period I spent in the group. Thanks Brian, Jee, Matt, Krit, and Ryan for the many brain-storming sessions and technical support. Thanks to Dr. Sek Chai, Dr. Peter Sassone, Dr Cory Hawkins, Dr. Hongkyu Kim, and Dr. Jongmyon Kim for technical support and encouragement.

Thanks to Priya Benjamin for being a steady source of inspiration. Thanks for helping navigate through the difficult times during this academic journey.

Finally, I would like to dedicate this thesis to my family for years of support and encouragement. I am grateful to my parents Mr. Michael Apewokin and Mrs. Esther Apewokin for providing me with an environment where my curiosity was encouraged. I am also grateful to my brothers Dr. Senu Apewokin, Sena Apewokin and Edem Apewokin for challenging me to reach further.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The combination of low-cost imaging chips and high-performance, multicore, embedded processors heralds a new era in portable vision systems. Early vision algorithms have the potential for highly data-parallel, integer execution. However, an implementation must operate within the constraints of embedded systems including low clock rate, low-power operation with limited memory. This dissertation explores new approaches to adapt novel pixel-based vision algorithms for tomorrow's multicore embedded processors. It presents:

- An adaptive, multimodal background modeling technique called *Multimodal Mean* that achieves high accuracy and frame rate performance with limited memory and a slow-clock, energy-efficient, integer processing core.
- A new workload partitioning technique to optimize the execution of early vision algorithms on multi-core systems.
- A novel data transfer technique called *cat-tail DMA* that provides globally-ordered, non-blocking data transfers on a multicore system.

By using efficient data representations, *Multimodal Mean* provides comparable accuracy to the widely used Mixture of Gaussians (MoG) multimodal method. However, it achieves a 6.2x improvement in performance while using 18% less storage than MoG while executing on a representative embedded platform.

When this algorithm is adapted to a multicore execution environment, the new workload partitioning technique demonstrates an improvement in execution times of 25%

with a 125 ms system reaction time. It also reduced the overall number of data transfers by 50%.

Finally, the *cat-tail DMA* technique reduces the data-transfer latency between execution cores and main memory by 32.8% over the baseline technique when executing *Multimodal Mean*. This technique concurrently performs data transfers with code execution on individual cores, while maintaining global ordering through low-overhead scheduling to prevent collisions.

CHAPTER 1

INTRODUCTION

1.1. Motivation

Demand for portable, low-cost, high computation platforms for multimedia and telecommunications applications is driving today's and tomorrow's embedded systems. Examples range from small systems, such as cellular phones, PDAs, and gaming consoles, to large, distributed systems, such as multi-node video surveillance systems. Regardless of the application, embedded computing systems are subject to more rigid cost, size, power, thermal, and real-time performance constraints than traditional general-purpose computing systems.

With the development of low cost embedded imagers, there is an opportunity to integrate early vision algorithms with real-time embedded systems. For example, the separation of salient foreground objects from uninteresting background is necessary in important applications such as vehicle collision avoidance, pedestrian tracking, anti-terrorist surveillance, and autonomous vehicle control. These applications demand real-time execution in a partially or fully embedded system (e.g. on a moving vehicle).

Embedded systems for real-time execution of early vision algorithms present unique demands. These algorithms require the transfer of large amounts of data between the execution units (where the images are processed) and off-chip memory (where the images are stored). A wide, high-frequency bus is desirable to support the transfer of high-bandwidth data but this is not typically affordable on an embedded platform. Also, a large storage area is needed on the computing cores to process data in larger blocks and reduce the number of transactions. With advances in process technology and the availability of billions of transistors, larger on-chip memory and ever-improving high-speed bus structures are staple features on general-purpose processors and will only improve for future generations. However, embedded platforms will be unable to follow the same trend due to embedded design constraints such as low-power. Keeping data readily available for execution on computation cores presents a key challenge.

Early vision workloads are demanding in terms of both memory and computational requirements. They typically involve extracting context out of a large quantity of pixel data and having the data readily available on the execution cores is important for real-time operation. Furthermore, the sheer volume of operations puts additional constraints on the computational requirements. For example, adding a single operation per pixel to a given algorithm is magnified by the large number of pixels and this has direct impact on real-time performance. Similarly, adding a single integer field to an image reference model is magnified across the image. Redesigning early vision algorithms to be more efficient in terms of storage and computation will significantly affect their ability to execute in real-time on embedded platforms.

Unlike general-purpose workloads however, early vision workloads are streaming in nature and feature very little data reuse. Also, the program control characteristics and data access patterns of these workloads are very predictable. This means the traditional solutions employed by general-purpose architects, such as hardware caching and pre-fetching to reduce memory latency as well as speculative execution and branch prediction to increase throughput, are not transferable to vision platforms. However, the same memory and throughput concerns remain for both workloads and new techniques to address those for early vision workloads are important for designing high performance systems.

Multicore processing is the future of high performance desk and laptop architectures (e.g., Intel, AMD processor roadmaps (see PC magazine summary [58])). Embedded processor development traditionally follows with low power platforms (e.g., ARM Cortex A8 [59]). These architectures are well-suited for early-vision algorithms because they provide hardware support for concurrent execution of these highly data-parallel workloads. However, caching, speculative execution, branch prediction, and other latency-reduction techniques that are commonly employed in general-purpose multicore processors are ill-suited for embedded vision applications. To achieve efficient, real-time execution of high-bandwidth early vision algorithms, architects must apply concurrent exploration of both architectural and algorithmic optimizations to system design.

Thesis Objectives: This dissertation explores mapping early vision algorithms onto multicore embedded platforms with emphasis on high performance and efficiency. Specifically, it uses the design of a pedestrian-tracking system as a representative case and addresses challenges with respect to the following:

- The effect of the modeling and tracking choices employed in the design of pedestrian-tracking software applications on the memory and computational requirements of embedded platforms and the implications for real-time performance.

- The effects of transferring large amounts of data between memory and computation cores when executing early vision algorithms on multicore embedded platforms and techniques to optimize the work done per transfer.
- The impact of data transfer memory latency on real-time performance of early-vision algorithms on multicore embedded systems and optimizations to minimize the transfer latency without sacrificing utilization of core local storage area.

The identified issues can have significant impact on the real-time performance of an embedded pedestrian-tracking system.

1.1.1. Memory and Computation Challenges of Early Vision Algorithms on Embedded Systems

Although several pedestrian-tracking applications have been proposed, they traditionally targeted desktop execution platforms and therefore modeling and tracking were approached accordingly. Real-time performance in real-world environments was not the main focus of the algorithms and therefore the implementation costs and complexity of the algorithms were not key design factors. Also, because they typically run above the operating system, they are unable to exploit the new multicore general-purpose processor designs. New approaches are required to achieve real-time performance on multicore embedded platforms.

Several techniques exist for modeling objects for tracking, such as color histograms, and shape analysis. The actual tracking algorithms also feature varying degrees of complexity and heavily employ floating-point computation. As a result, the techniques are usually computationally and memory intensive and in stable environments the accuracy improves with more complex modeling. However, in dynamic, real-world environments complex modeling does not necessarily yield improved accuracy. A thorough analysis of the individual components of video surveillance applications is necessary to identify which areas could be redesigned to significantly improve overall performance for real-world embedded systems.

A study of video surveillance workloads reveals that a significant portion (up to 95 %) of the execution time for this class of workloads involves running early vision algorithms, such as background modeling [21]. These memory intensive (and potentially computation intensive) workloads are particularly challenging for embedded platforms because they have limited on-chip memory and reduced computational capabilities. Limited on-chip memory has a direct impact on the performance of early vision algorithms on embedded systems because a given frame is processed in blocks constrained in size by the amount of data the processing core can accommodate. Furthermore, the modeling and data structuring choices made when representing a pixel in a given algorithm directly affects the block-size. The smaller the block size for the algorithm, the more iterations are required to completely process a given frame. As a result, the execution time is inversely proportional to the block size.

The first portion of this thesis provides a framework for designing accurate, high-performance, pedestrian-tracking application software for embedded systems. It combines effective, inexpensive object modeling and tracking with fast, adaptive, and accurate background modeling to achieve high-performance without sacrificing accuracy.

1.1.2. Processing and Data Bandwidth Challenges on Multicore Embedded Systems

Early vision algorithms that have in the past been developed for uniprocessor platforms must be redesigned for distribution and concurrent execution on the individual cores of a multicore embedded platform. Since the algorithms are highly data-parallel, a naïve solution will be to partition data (each image frame) into equal parts, each of which is transferred to a computing core for execution. This process can be repeated to complete the entire video workload. However, this solution does not yield optimal performance and alternate techniques are required for improving the performance.

1.1.3. Memory Transfer Latency Challenges on Multicore Embedded Systems

Early vision workloads are highly data-parallel and feature little data reuse. Existing hardware techniques for reducing memory latency, such as the use of caches and other pre-fetching mechanisms, are ill-suited for these workloads. As a result, embedded multicore processors typically feature DMA-based data transfers and no hardware caches. This feature can result in tremendous performance improvements but also presents significant software design challenges because data transfers are moved to the domain of the application programmer. Also, there is inherent difficulty and complexity in designing efficient parallel programs that can fully exploit multicore hardware resources on embedded systems.

Efficient data transfer between main memory and execution cores is particularly important for multicore embedded systems because they have much smaller local storage areas as compared to multicore general-purpose processors. As a result they require several more iterations to completely process a single image frame. This increases the frequency of data exchanges and the potential for collisions on a multicore system, which in turn can result in high data transfer latency and potentially costly memory bottlenecks. This work also explores techniques to minimize the data transfer latency between main memory and execution cores while optimizing utilization of local storage on embedded multicore systems.

1.2. Problem Statement and Research Contributions

The purpose of this research is to efficiently map early vision algorithms onto multicore embedded platforms to achieve high-performance execution of applications without sacrificing accuracy. Pedestrian-tracking is used as a representative workload for three thesis contributions:

1. A video surveillance software development framework that minimizes computational and storage requirements on embedded systems by using efficient object modeling and tracking techniques supported by a fast, accurate, adaptive background modeling algorithm.
2. A workload organization and processing technique that enhances the performance of early vision algorithms on multicore embedded platforms by optimizing algorithm execution on computing cores and minimizing the number of data transfers required for program execution.
3. A technique to minimize memory latency of image transfers on multicore embedded systems by performing transparent, global DMA-scheduling with

concurrent program execution while simultaneously ensuring high-utilization of core local storage areas.

1.3. Research Approach Summary

The research in this dissertation is approached from a systems perspective by addressing both software and hardware components. Concurrently addressing both the architectural and algorithmic challenges presented in Section 1.2 is necessary to design efficient, high-performance embedded systems. From the algorithms perspective, the research focuses on the development and implementation of efficient execution early vision algorithms. Because early vision workloads constitute a significant portion of the overall workload [21], improvements to these algorithms will yield significant overall program speedup and this dissertation first focuses on improving their performance on embedded systems.

A new early vision background modeling algorithm that features lower computational and storage costs on embedded platforms is presented. This adaptive background modeling technique, called *Multimodal Mean*, is evaluated against several existing background modeling techniques on several representative embedded platforms. The evaluation compares this algorithm with several existing pixel-level background modeling techniques in terms of their computation and storage requirements, and functional accuracy for representative real-world video sequences, across a range of processing and parallelization configurations. The *Mulitmodal Mean* technique provides the accuracy of the most popular of the multimodal algorithms (Mixture of Gaussians

[23]) algorithm while executing at frame rates comparable to other less expensive techniques.

This approach is extended to object modeling and tracking in the video surveillance applications. This dissertation introduces a framework composed of an inexpensive kinetic modeling and tracking of objects supported by a fast, accurate background model for designing surveillance applications. By using this approach, the surveillance applications feature comparable accuracy to other techniques while achieving high frame rate execution on embedded computing platforms.

The architectural perspective of this research focuses on optimizations that allow early vision algorithms to run efficiently on multicore embedded systems. It explores optimizations with respect to data storage, data transfer latency, and data reuse. As described previously, the availability of local storage on execution cores is limited, a problem which is not as dire on multicore general-purpose platforms. As a result, optimizations that allow the reuse of the background model for processing multiple blocks of an image significantly improve performance. This dissertation presents such an approach and shows how several configurations impact the overall execution time.

Finally, the dissertation addresses data transfer efficiency on multicore embedded systems. The approach is to reduce the memory transfer latency on multicore embedded systems to ensure high utilization of computing cores. It presents a technique that leverages the available hardware to perform concurrent data transfer and program execution and minimize latency. Furthermore, it provides globally ordered data transfers among cores to prevent collisions and improve efficiency. Finally, it maintains high utilization of core local storage area while performing concurrent execution.

1.4. Results Summary

These results of this dissertation can be summarized as follows:

- An object modeling and tracking framework [16], [18], and [19] is presented that achieves 92% tracking accuracy for pedestrian-tracking applications and operates at 0.78 fps when processing 640x480 pixel images on a representative embedded platform.
- A background modeling algorithm [17], [30] is introduced that provides comparable image quality and accuracy to the Mixture of Gaussians (MoG) algorithm with the performance of other more efficient but less accurate background modeling techniques. This algorithm executes 6.2× faster than MoG on a representative embedded platform and 4.23× faster on a more capable platform. It also requires 18% less storage per pixel than MoG and uses only integer operations
- A workload partitioning technique [52] is described that optimizes the execution of background modeling algorithms on multicore systems. The technique results in a 25% increase in processing frame rates when executing *Multimodal Mean*, and a 50% reduction in the number of image transfers. It also features little overhead (0.023%) in image decoding times and an overall system delay of 0.125s for 320x240 frames.

- A DMA-transfer technique is presented called *cat-tail dma* that provides globally-ordered, non-blocking DMA transfers on a multicore system. Using this technique, data transfers between main memory and processing cores are reduced by 32.8% for *Multimodal Mean*. Also utilization of core local storage is improved by 60% over other buffering/processing techniques.

1.5. Overview of Content

This thesis is organized as follows: Chapter 2 introduces a modeling and tracking framework for pedestrian-tracking on embedded systems that minimizes memory storage and computation requirements. This chapter examines the components of a pedestrian-tracking workload and identifies a framework for enhancing application performance without sacrificing accuracy. Chapter 3 describes the background model that supports the pedestrian-tracking framework by providing fast, accurate background modeling on embedded systems. Chapter 4 describes a technique to optimize the processing of background modeling workloads on multicore embedded systems. It introduces a workload partitioning/optimization scheme that optimizes performance on multicore systems. Chapter 5 presents a technique for efficient transfer of images between main memory and the computing cores on a multicore system. The conclusion and future work are presented in Chapter 6.

CHAPTER 2

PEDESTRIAN-TRACKING APPLICATION

2.1 Introduction

Tracking pedestrians in a dynamic scene is challenging for several reasons. People change shape as they move and several blobs may exhibit the same general shape, making one-to-one correspondence difficult. Also, frequent occlusion, merging of individual people into groups, and splitting into individuals again makes tracking complicated. In addition, pedestrians frequently make path adjustments to avoid collisions that can result in fluctuations in their walking speeds over a large number of observed frames.

There is also an increasing desire to perform pedestrian-tracking on embedded platforms attached to imagers that form sensing nodes. These nodes, which may be part of a broader surveillance system, must be cheap and power-efficient to make the entire system feasible. More importantly, the nodes must be able to run the pedestrian-tracking algorithms accurately and in real-time. Figure 1 illustrates such a system made up of an embedded platform and a webcam connected through a wireless network to a central server. Each node has a distinct field of view and the server performs extra processing to aggregate and analyze the information from each node.

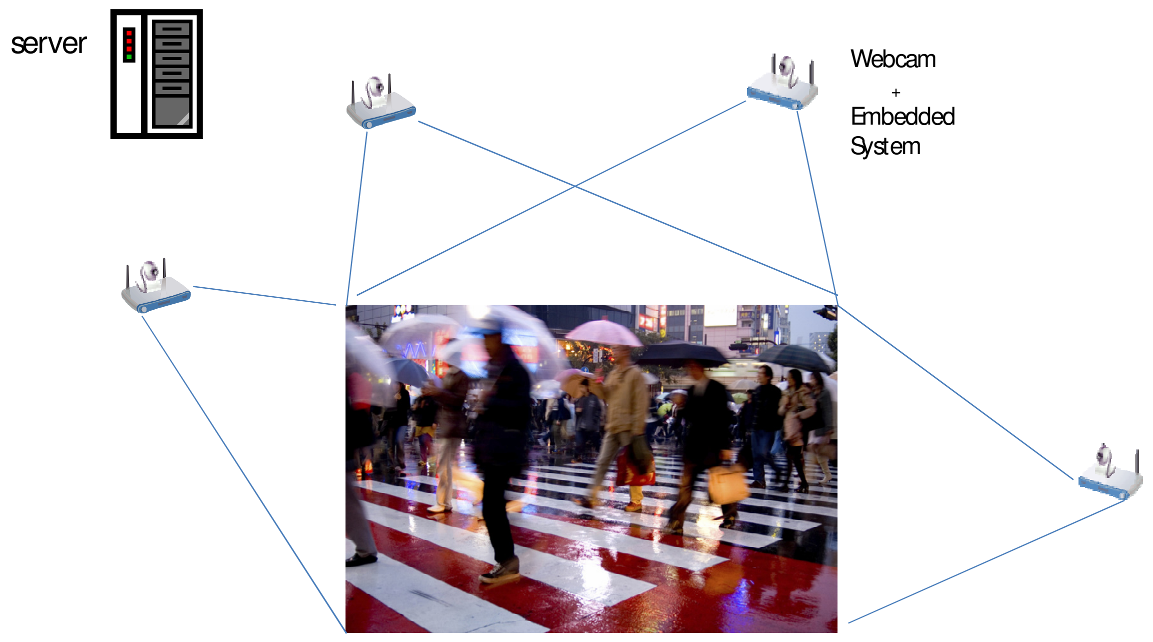


Figure 1 Video surveillance system

The goal of the work in this chapter is to develop an accurate pedestrian-tracking framework that supports real-time performance of software applications on embedded targets. It begins by describing the key challenges facing the design of pedestrian-tracking applications in this environment. Then an evaluation of several modeling and tracking approaches is performed to highlight the advantages and disadvantages of the approaches. Based on the analysis, a new object modeling and tracking approach is presented for embedded systems.

The algorithm is targeted for embedded systems and reduces computational and storage costs by using an inexpensive kinematic tracking model with only fixed-point arithmetic representations. It leverages from the observation that pedestrians in a dynamic scene tend to move with uniform speed over a small number of consecutive frames. As a result, if foreground objects are clearly identified, they can be tracked with high accuracy over a short distance. The pedestrian-tracking application is built on an integral of such incremental tracking over a long period. Accommodations are made for confusing behavior such as occlusions, merging, and splitting. An accurate, multimodal background modeling technique is used to segment the foreground (moving people) from the background. This component offers important support to the framework by providing accurate segmentation while minimizing processing and storage costs. A connectivity analysis step is performed concurrently with the background modeling and is used to identify blobs in the foreground and calculate the center of mass of each blob. Finally, correspondence is established between the center of mass of each blob in the temporally closely-spaced frames. The algorithm is evaluated on a real outdoor video sequence taken with an inexpensive webcam and the implementation successfully tracks each pedestrian from frame to frame in real-time. The algorithm performs well in challenging situations resulting from occlusion and crowded conditions, and achieves real-time performance on an actual embedded system.

2.2 Current People Tracking Approaches

The literature on pedestrian-tracking techniques is extensive and covers a broad range of applications. Yilmaz, Javed, and Shah conduct a general survey of object tracking, including articulated object trackers that apply to person tracking [1]. Recent surveys focused on using articulated object models of human kinematics in particular have been provided by Aggarwal and Cai [2], Gavrila [3], and Moeslund and Granum [4]. One large class of applications involves tracking the movement of a few people in a sparsely populated scene. The goal is to automatically infer the particular activity being performed in a given scene. Pfinder [5] uses a multi-class statistical model of color and shape to represent a person in various positions in a scene. After an initialization period in which several representations are obtained, it is able to interpret the action of the individual while updating the model to incorporate new actions. The W4 algorithm [6] completely ignores color and uses a combination of shape analysis and tracking to locate people and their parts. In [7], a person detector is used to locate a person's limbs and a discriminative appearance model is built over a given number of frames for each limb. Tracking is done by detecting the collective appearance model in each frame. Finally, a tracking-based event detection CCVT system is described in [8]. With this approach certain blob and scene basic characteristics such as blob positions, blob speed, and people density are extracted from the foreground frame. These parameters are compared with semantic descriptions of prior events for classification. The techniques described above work well in scenarios where there are very few foreground objects and where the objects (people) make up a significant portion of the scene making individual features (e.g. limbs) discernible. These systems have been implemented and reported to perform with reasonable accuracy.

At the other end of the spectrum, another large class of applications involves following the trajectory of multiple people as they move through a scene. In some situations where there is heavy pedestrian traffic there is less feature detail and tracking multiple people presents new challenges. In [9] a multiple-people-tracker is proposed that uses a stochastic approach based on the evaluation of the maximum a posteriori probability (MAP). A state history vector is maintained that contains records of the position, velocity, acceleration, dimension, and unique identification tag of all blobs detected in a given frame at time t. An observation state vector is also kept that records the blob states observed up until frame t. Trajectory tracking is performed by computing the configuration sequence that maximizes the a posteriori probability distribution over states conditioned on observations. In [10] a Markov chain Monte Carlo (MCMC)-based method is used to calculate the MAP for establishing correspondence but objects are modeled based on shape and color. The authors also present a detailed description of optimizing the computation of the MAP both for single-object and multiple-object situations. Similarly, MCMC is used to perform monocular 3D human tracking in [11]. Particle filter based techniques such as [12], [13], and [14] generate multiple predictions based on a dynamic model from which a likelihood function is used to determine correspondence.

The techniques described above have been demonstrated to be successful in experiments on PC platforms. However, their real-time performance capabilities when implemented on resource-constrained embedded platforms will be challenged due to computation and storage limitations. For the MAP techniques, expensive floating-point calculations are required to achieve the desired accuracy, and complex optimizations are often necessary. Also, color, shape, and appearance modeling as well as memory storage using such representations can be expensive for embedded platforms. Alternative kinematic-based algorithms have been proposed [11] that model the velocity and acceleration of blobs over a long number of frames. However, this information can be difficult to model because of the interaction of pedestrians which causes blobs to merge and split.

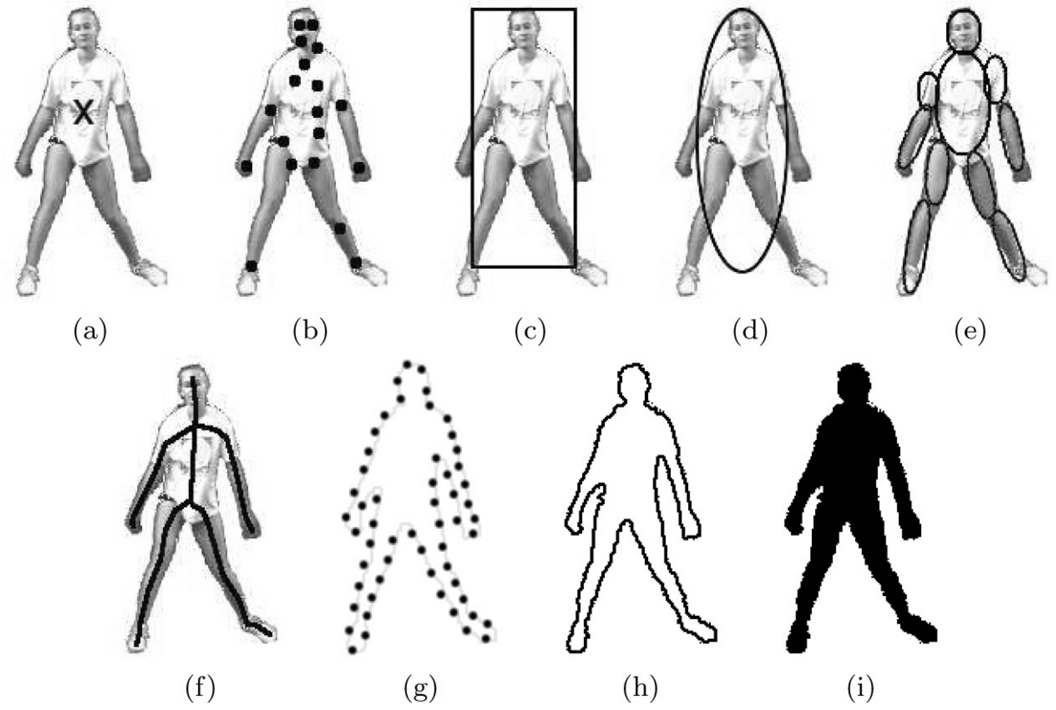
2.3 Object Modeling Rationale

Tracking by color is a popular technique for a variety of applications [1] but is ill-suited for pedestrian-tracking in a dynamic scene. Frequent occlusion can make color information in a particular pedestrian blob inconsistent as a pedestrian traverses the scene. Also, there are shadowing effects due to other pedestrians and illumination changes as a pedestrian blob moves to better lit areas of the scene. These effects lessen the accuracy of the histogram matching techniques employed for color tracking.

Similarly, tracking by shape suffers from the occlusion problem. Also, pedestrian blob shapes are neither unique enough between frames nor consistent enough across frames for accurate tracking.

Another problem with using color and shape for object modeling is the cost in terms of both computation and storage. Color histogram-based techniques model objects by creating a histogram using groups of similar pixels in the object and this process can be computationally expensive. The same problems exist when using shape analysis because pedestrian shapes are generally more complicated to represent than geometric shapes and hence more expensive to model.

Considering the high costs and unpredictable accuracy this work avoids using color or shape modeling for real-time pedestrian-tracking applications. Instead, it proposes using a kinematic model based on pedestrian positions that completely discards color and shape information after foreground extraction. Pedestrians are modeled as a centroid as shown in Figure 2a.



Source: Yilmaz et al "Object Tracking: A Survey," *ACM Computing Surveys,* Vol. 38

Figure 2 Object representations
(a) Centroid, (b) multiple points,(c)rectangular patch, (d) elliptical patch, (e) part-based multiple patches, (f) object skeleton, (g) complete object contour, (h) control points on object contour, (i) object silhouette.

The tracking algorithm leverages from the observation that pedestrians in a dynamic scene tend to move with uniform speed over a small number of consecutive frames. As a result if foreground objects are clearly identified, they can be tracked with high accuracy over a very short distance. The pedestrian-tracking application is built on an integral of such incremental tracking over a long period.

To achieve high accuracy using the kinematic model described above, foreground objects (people) must be clearly and consistently distinguishable. To support this kinematic approach, an accurate, adaptive background modeling algorithm, called multimodal mean [17] has been developed. This algorithm is introduced and discussed in further detail in Chapter 3.

After, accurately segmenting foreground objects, the people tracking algorithm uses center of mass information to model blobs as pedestrians. It uses separate models of pedestrian activity both in the short term (over a few consecutive frames) and long term (as the pedestrian traverses the scene) to perform tracking. An added benefit to this choice is that the tracking algorithm is robust and works in challenging conditions such as in poorly-illuminated environments or when using cheap, low-resolution imagers where color information is unreliable.

2.4 Pedestrian-Tracking Algorithm

The pedestrian-tracking algorithm uses accurate position information for both modeling and tracking of pedestrians. Multimodal mean [17], a fast, adaptive background modeling technique, is used for the foreground/background segmentation. A connectivity analysis step is performed within the segmentation procedure to group foreground regions of interest. Each column C_t in a given frame *t* is identified as a column of interest if

$$P_{c.t} \rangle C_{th} ,$$

where $P_{c.t}$ is the number of foreground pixels in C_t, and C_{th} is a fixed, predetermined threshold for the column pixel density of a pedestrian.

A pedestrian region is identified when

$$(R_{c.t} \rangle R_{th}) \cap (F_{r.t} \rangle F_{th}) ,$$

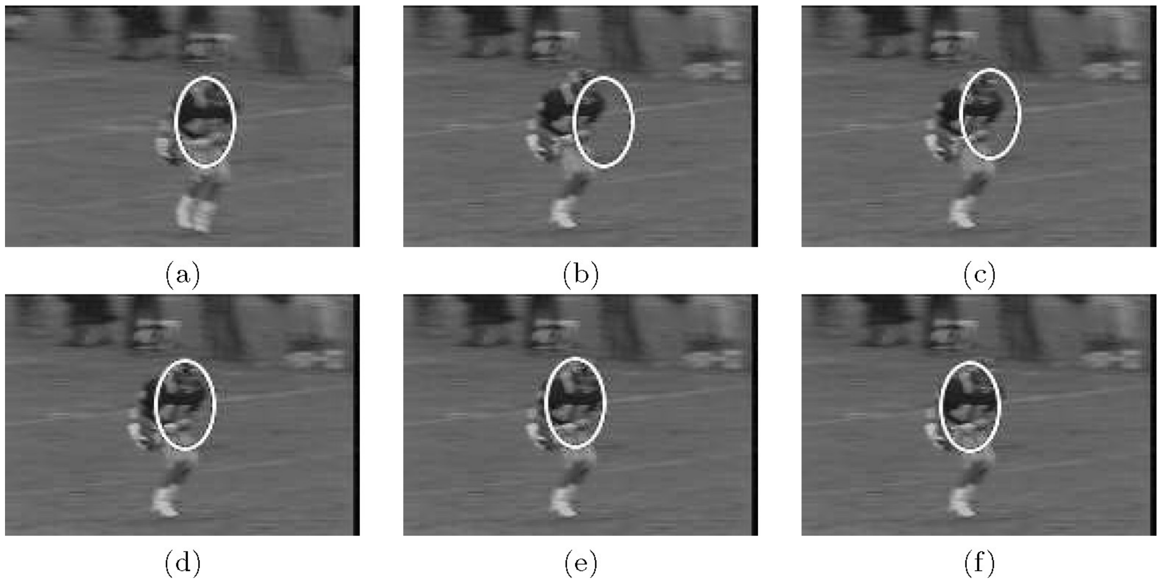
where $R_{c.t}$ is the number of adjacent columns of interest in frame *t*, and $F_{r.t}$ is the total number of foreground pixels contained in all those columns. R_{th} and F_{th} are predetermined thresholds for the minimum width of a pedestrian and the minimum pixel density of a pedestrian respectively.

After all independent blobs have been identified in each frame, the algorithm establishes correspondence with previously observed blobs to perform tracking. Each pedestrian blob P_t observed in frame *t* is modeled as a single point $(P_{t.x}, P_{t.y})$ which is the center of mass of the blob. It should be noted that two pedestrians whose positions overlap each other may be represented using a single blob. A frame vector V_t which contains each pedestrian blob *i* observed during frame t is maintained.

$$V_t = (P_{t.x.i}, P_{t.y.i})$$

Two history records are maintained for tracking pedestrians as they traverse the scene. One is the local history tracker and the other is the global history tracker. The local history tracker is used to follow the short term progress of pedestrians from frame to frame. This tracker uses a variation of the mean-shift tracking algorithm shown in Figure 3 to track short term progress of pedestrians. In the presence of occlusion or blob merges and splits, which are common patterns when observing pedestrians, the local history is unable to decipher the scenario because it only has an account of what happened in the most recent three frames. The global history tracker records additional details about all blobs over the entire period that they traverse the scene. This enables the tracking system to recover from blob occlusion or merges and splits.

By separating the long and short term history tracking information, the algorithm adapts to the changing kinetics of pedestrians without employing complex models.



Source: Yilmaz et al “Object Tracking: A Survey,” *ACM Computing Surveys,* Vol. 38

Figure 3 Mean-shift tracking iterations
(a) estimated object location at time t − 1, (b) frame at time t with initial location estimate using the previous object position, (c), (d), (e) location update using mean-shift iterations, (f) final object position at time t.

2.4.1 Local History Tracker: Dealing with Correspondence over Short Periods

The local history tracker records position information of pedestrians over three consecutive frames during which constant velocity and constant acceleration can be assumed. A blob P_t in the current frame *t* is a candidate for matching using the local history tracker if

$$\left|(P_{t.x}, P_{t.y}) - (P_{t-1.x}, P_{t-1.y})\right| \langle D_{th},$$

and $$((P_{t.x}) - (P_{t-1.x})) \times D_{rn} > 0$$

where P_{t-1} is a blob in the immediately preceding frame, *t-1*. D_{th} is chosen to be about 2x the average distance traversed by a pedestrian during the processing of the sequence. D_{rn} is the direction of travel for blob P_{t-1}, and is obtained from the global history record which is described further below. The local history record, L_t, is made up of the frame vectors from the three most recently observed frames *t-2*, *t-1*, and *t*.

$$L_t = (V_{t-2}, V_{t-1}, V_t).$$

During processing of the current frame, the algorithm first attempts to establish correspondence between candidate blobs and other blobs that have been successfully tracked from frame *t-2* to *t-1*. Assuming constant velocity for a given blob P_t between the three consecutive frames *t*, *t-1*, and *t-2*, the change in center of mass of a given blob between frame *t* and *t-1* is approximately equal to that between frame *t-1* and *t-2*.

Specifically $(P_{t.x}, P_{t.y}) - (P_{t-1.x}, P_{t-1.y}) \approx (P_{t-1.x}, P_{t-1.y}) - (P_{t-2.x}, P_{t-2.y})$.

The candidate blob P_t in frame *t* that minimizes the difference in change of center of mass is matched with the one successfully tracked between frame *t-1* and *t-2*.

$$Min\left(\left|\Delta_{(P_{t.x}, P_{t.y}),(P_{t-1.x}, P_{t-1.y})}\right| - \left|\Delta_{(P_{t-1.x}, P_{t-1.y}),(P_{t-2.x}, P_{t-2.y})}\right|\right)$$

If a tracked blob is occluded in either frame *t-1* or *t-2*, the matching attempt described above fails. In this case a match is made with a blob in frame *t-1* by selecting the candidate blob in *t* that minimizes the center of mass difference between the two blobs.

$$Min\left(\left|\Delta_{(P_{t.x}, P_{t.y}),(P_{t-1.x}, P_{t-1.y})}\right|\right).$$

2.4.2 Global History Tracker: Dealing with Occlusion, Merging, and Splits

The local history tracker fails during incidences of blob merges and splits, as well as during occlusions over a long number of consecutive frames. The global history tracker is used to address those problems, and is made up of a collection of records kept for each pedestrian blob within a specified observation window.

The global history tracker is used to match pedestrian blobs that are unmatched using local history. This tracker contains position and velocity information of pedestrians over its entire existence and is made up of four fields. The first field *Last Pos* is a tuple that records the last observed center of mass point for a given blob identified by field *ID*. The direction field, *Drn*, gives an indication of the general direction of motion of blob *ID*. It is initialized to zero when a new blob record is created and increased by one if the blob is matched to the right of its last position. If it is matched to the left the *Drn* value is decreased by one. The last field *Frame Num* records the frame number when blob *ID* was most recently observed.

The time elapsed from the last update of a given blob is computed by taking the difference between the current frame and the contents of the *Frame Num* field recorded in the blob. Similarly, the change in position as well as the direction of change is computed from the current unmatched blob position and the last position recorded in the blob. A given blob P_t in the current frame, is a candidate match with an entry G_t in the global history tracker if

$$\left|\left(P_{t.x}, P_{t.y}\right) - \left(G_{t.x}, G_{t.y}\right)\right| \langle G_{th},$$

and
$$\left(\left(P_{t.x}\right) - \left(G_{t.x}\right)\right) \times D_{rn} > 0.$$

If there is more than one candidate blob, a match is determined by selecting the candidate blob that minimizes the difference in center of mass.

$$Min\left(\left|\Delta_{(P_{t.x},P_{t.y}),(G_{t.x},G_{t.y})}\right|\right)$$

The global history tracker is routinely maintained to prevent matching of blobs using stale history and also to prevent the storage structure from overflowing. When a pedestrian blob in the current frame is matched, the *Last Pos, Drn*, and *Frame Num* fields in the global history tracker are updated. Blob records are removed from the global history tracker under the following conditions:

$$CurrentFrame - FrameNum \langle O_{th} \text{ and}$$

$$((LastPosition \langle W_{th}) \,||\, (LastPosition \rangle (Width - W_{th})) \,\&\&\, (CurrentFrame - FrameNum) \rangle \langle E_{th})$$

O_{th} is a frame observation threshold which represents how long a blob can be considered occluded before it is considered lost and discarded. W_{th} is a width window at the beginning and end of each frame for which blobs may be leaving or entering the scene.

By separating the long and short term history tracking information, the algorithm adapts to changing kinetics of pedestrians without employing complex models. Also, by giving precedence to the local history tracker, it finds the closest match for a given blob during regular operation, and has a recovery system for scenarios where there is a deviation from normal behavior e.g. occlusion. Most importantly, keeping the models separate provides the option of modeling different scenarios while minimizing the complexity.

2.5 Experiment and Analysis

The tracking algorithm was evaluated on an outdoor sequence taken with an inexpensive webcam. The scene involves a busy walkway outlined by trees on a sunny day. Under those real-world conditions, waving trees and shadows could result in a noisy background and could affect the segmentation. The video was recorded at 30 frames per second (fps) and down sampled to 1 fps for processing, and 200 frames where processed for the experiment. At this reduced rate there was measurable change in pedestrian locations from frame to frame. Processing was performed at a relatively high resolution, with the size of each image at 640x480 pixels. The tracking algorithm was implemented in C and compiled using Microsoft Visual Studio 2005 for Windows CE 6.0 embedded.

The execution platform was an eBox-2300 Thin Client VESA PC running Windows Embedded CE 6.0 [31], [20]. This was chosen as a baseline platform for evaluating the pedestrian-tracking application and featured very modest memory and processing specifications. The eBox, shown in Figure 2, incorporates a fanless Vortex86 SoC (includes a 200MHz x86 processor that dissipates < 3 Watts) plus 128MB SDRAM (PC133), three USB ports, a 10/100 Ethernet port, and a compact flash slot. The processor is an integrated version of the Pentium processor which was originally introduced commercially in 1993 (about 15 years ago). The platform is 11.5 × 11.5 × 3.5 cm in size, weighs 505g, and is designed for low power operation. Because of its limited 128MB internal memory, a customized lightweight kernel occupying approximately 19MB was constructed. Image sequences were downloaded prior to each evaluation run.

Figure 4 eBox Vesa PC

Sequence A shows a long sequence taken from the results where tracking is performed over a 24s period. The frames are shown at 3 sec intervals. The sequence shows that the algorithm correctly tracks the pedestrians as they interact and generate some interesting scenarios while walking across the scene. Pedestrians 1 and 2 are traveling at different speeds and begin as independent blobs in Frame 68 till they are merged into a single blob in Frame 77. Pedestrian 4 was initially traveling behind pedestrian 2 and then stops walking. She is tracked for a while as foreground and enters the background until she starts walking again. Also, there are several instances involving occlusion where pedestrians traveling in opposite directions merge into a single blob, and separate later. In these scenarios the blobs are correctly tracked before and after the occlusion events.

Sequence B shows results from a very crowded period of the video. Pedestrian 7 and her companion (from Sequence A) have become almost stationary and are having a conversation and therefore are now part of the background. They have remained in the video while a completely new set of pedestrians walk across the scene. Pedestrians 4, 8, and 9 are tracked among the crowd from frames 150 to 158. Again, the occlusions involving pedestrians 4, 5, and 1 are handled correctly.

A ground truth was created to evaluate the accuracy of the algorithm. This process involved manually observing each pedestrian as they traversed the scene and comparing the matches produced by the algorithm with those matched by eye. All mismatches were considered inaccuracies and the program was penalized. Also, identification of blobs where no actual blobs existed, and failure to identify fully autonomous blobs (where there was no contact with another blob) were considered inaccuracies and incurred penalties. A window of 50 pixels was maintained at the beginning and end of each frame where tracking was ignored to allow pedestrians to leave and enter the scene. Using these criteria the algorithm achieved an accuracy of 92% for this sequence

One challenging scenario for the algorithm involved the merging and splitting of blobs in the same direction. In rare instances where the difference in pedestrian speeds was large, the pedestrian identities were switched. This can be resolved by reducing the down sampling rate. Also, in some instances where blobs were not fully formed due to noise, the algorithm missed blobs in some frames. It usually recovered in the next frame when the segmentation was cleaner.

The performance of the algorithm on the eBox was also measured. The algorithm ran at 0.78 fps when processing 640x480 pixel images. This frame rate is appropriate for tracking pedestrians walking in real-time, so the resolution at which the images were processed was increased to achieve better accuracy. Considering the platform, this frame rate and resolution is very suitable for real-time, pedestrian-tracking. Run-time memory usage averaged about 33 MB which constituted about 25% of what was available on the eBox.

This chapter introduced a framework for designing real-time, pedestrian-tracking software for embedded systems. By using inexpensive tracking and object modeling representations the memory and computation requirements of the application are significantly reduced. However, to preserve accuracy, a fast, multimodal background modeling algorithm is needed to provide the accurate segmentation on which the kinetic tracking model can rely. The algorithm must also be designed with careful consideration to memory and computation costs because it will execute on an embedded platform. The next chapter presents the *Multimodal Mean* algorithm which satisfies those requirements, and is used to support the framework.

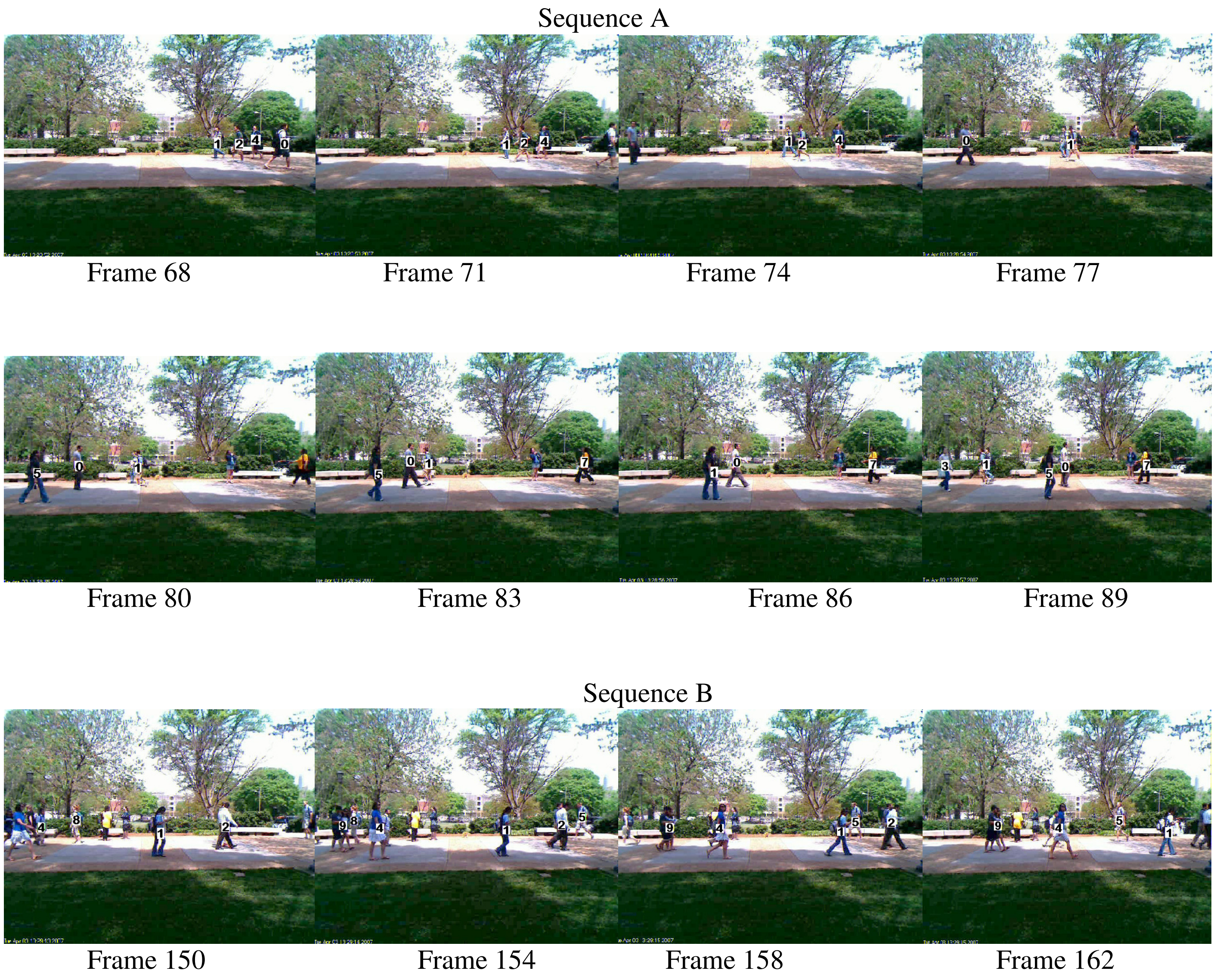


Figure 5 People tracking result

CHAPTER 3

MULTIMODAL MEAN BACKGROUND MODELING TECHNIQUE

3.1 Introduction

Techniques for automated video surveillance utilize robust background modeling algorithms to identify salient foreground objects. Typically, the current video frame is compared against a background model representing elements of the scene that are stationary or changing in uninteresting ways (e.g. rippling water or swaying branches). The 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 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.

This chapter introduces a new adaptive technique, *Multimodal Mean* (MM), which balances accuracy, performance, and efficiency to meet embedded system requirements. *Multimodal Mean* models each background pixel as a set of up to K modes, each represented as a running average pixel value in a structure called a cell. Each cell consists of running averages for each color component in a three-component color representation such as RGB or HSI.

Multimodal Mean was evaluated against several representative pixel-based background modeling techniques on a real embedded platform using data from a real-time embedded environment. The techniques were evaluated with respect to computational cost, storage, and extracted foreground accuracy. The techniques ranged from simple, computationally inexpensive methods, such as frame differencing and mean/median temporal filters [22], to more complex methods, such as the multimodal Mixture of Gaussians (MoG) [23] approach.

Commercial-of-the-shelf components were employed to build a low-cost, low-power, and portable embedded platform to serve as the testbed for the evaluation. The results demonstrated that the proposed MM algorithm achieved competitive real-time foreground accuracy under a variety of outdoor and indoor conditions with the limited computation and storage of a low-cost embedded platform. More specifically, *Multimodal Mean* technique achieved accuracy comparable to multimodal MoG techniques but with a significantly lower execution time.

3.2 The Case for a Fast Adaptive Background Model

Chen et al [21] present a comprehensive analysis of computer vision workloads. They chose video surveillance as a representative case study of a complex computer vision application and profiled it with the Intel VTune Performance Analyzer. Their results showed that foreground/background segmentation was the most expensive module in the workload and accounted for up to 95% of the execution time. According to their analyzer, their background modeling algorithm consumed 1 billion micro-instructions for a frame size of 720x576 pixels and took 0.4s to execute on a 3.2 GHz Intel Pentium 4 processor. Further analysis of the module showed that about 60% of the background modeling computation time was 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. This task can be approached from two directions:

I. Optimizing background modeling algorithms for embedded systems, and

II. Identifying suitable execution platforms and optimizing processing and partitioning of background modeling data for those systems.

This chapter addresses the first task by introducing a fast adaptive background modeling algorithm targeted for embedded systems. The next chapter tackles the second task and explores techniques to optimize the performance of the new algorithm on a suitable embedded platform.

3.3 Related Background Modeling Work

A variety of techniques exist for background subtraction; see [22], [24], and [25] 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 [26] and cannot handle dynamic background elements, such as swaying tree branches.

Sliding window-based (or *non-recursive* [22]) 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 technique is more memory intensive as it requires *w* image frames of storage per processed image.

Recursive techniques [22] utilize only the current frame and parametric information accumulated from previous frames to separate background and foreground objects. They typically employ weighted means or approximated medians and require significantly less memory than the sliding window techniques. An *approximated median* algorithm is shown in [27] where the background is initialized by declaring the first image frame as the median. When a new video frame is acquired, the current image's pixel values are compared with those of the approximated median's 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 [27]. In [28] and [5], a *weighted mean* is used, whereby a percentage of the background pixel is used in combination with 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) [23] and Wallflower [26] 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 [23]

Wallflower [26] 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 source being switched on or off.

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

In this chapter, a new background modeling technique [30] is proposed that has the multimodal modeling capabilities of MoG but at significantly reduced storage and computational cost. A related approach [29] implements multimodal background modeling 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 MoG and [29], the proposed technique uses a linear parameter updating scheme as opposed to nonlinear updates of weights and pixel values, and it makes use of information about the recency of background pixel matches. Updating the background model in this manner allows for efficient storage of a pixel's long-term history.

3.4 Multimodal Mean Algorithm

Multimodal Mean 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 using 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., $I_{t.red}$ 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 the embedded implementation, K = 4 cells was chosen and the RGB color representation was used. 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:

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

where T_{FG} is a small threshold indicating the 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 $t \bmod d = 0$, 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. (In the experiments reported later in the chapter, the decimation rate d = 400, so decimation does not come into play in the test sequences. However, it is necessary for longer-term adaptation.)

When a pixel I_t does not match cells at that pixel position, it is declared to be 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:

$$r_{i,t} = \begin{cases} (& \text{when t mod w = 0} \\ \mathit{ı} & \text{when Bi matches It} \end{cases}$$

and t mod w $\neq$ 0

$$s_{i,t} = \begin{cases} \text{ı} & \text{when t mod w = 0} \\ \text{.} & \text{otherwise.} \end{cases}$$

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 set 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 the experiments, w = 32 was chosen.)

3.5 Evaluation on Embedded Platform

Several background modeling techniques were evaluated using three representative test sequences executing on an embedded execution platform. Each technique was 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 included:

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

The test suite includes two standard test sequences and a longer outdoor sequence captured using an inexpensive webcam (see **Table 1**). All sequences have a frame size of 160×120.

Table 1 Test sequences

Sequence	# Frames	Sampled Frame
Waving Tree	281	247
Bootstrapping	1000	299
Outdoors	201	190

The standard sequences, “Waving Tree” and “Bootstrapping,” are from the Wallflower benchmarks [26] and use the same sampled frame and associated ground truth. They contain difficult challenges for background modeling algorithms. Waving Tree contains dynamic background in the form of a wind-blown tree with swaying branches and leaves. Bootstrapping lacks a “foreground free” preamble for construction of the initial background model. This requires learning the background in the presence of continually changing foreground. These sequences are choreographed to present specific background modeling problems. A longer sequence with dynamic background and the continuous presence of foreground objects was also collected. This sequence contains an outdoor scene with varying illumination, moving trees, and subjects moving in varying patterns and positions. It was captured at 640×480 resolution at one frame per second. Afterward, the sequence was resized to 160×120 and a sample frame and ground truth was manually derived.

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 4 for comparable memory requirements.

Table 2 Algorithm Parameters

Algorithm	**Parameters**
Mean/Median (SW)	$\lvert window \rvert = 4$
Weighted Mean	α=0.1 for ut = (1-α)*ut-1 + αxt
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, Ex = 30 for x∈{R, G, B},
	TFG = 3, d = 400, w = 32

The execution platform used for the evaluation was the eBox-2300 Thin Client VESA PC running Windows Embedded CE 6.0 from Chapter 2. Each background modeling technique was implemented in C and compiled for Windows CE using Microsoft Studio. Algorithm data storage was limited to 40MB. This affected the variable window size for the sliding window techniques and the number of modes for the multimodal techniques.

3.5 Results

The accuracy and image quality of each method is compared in Figure 6**,** Figure 7, Figure 8, and Figure 9.

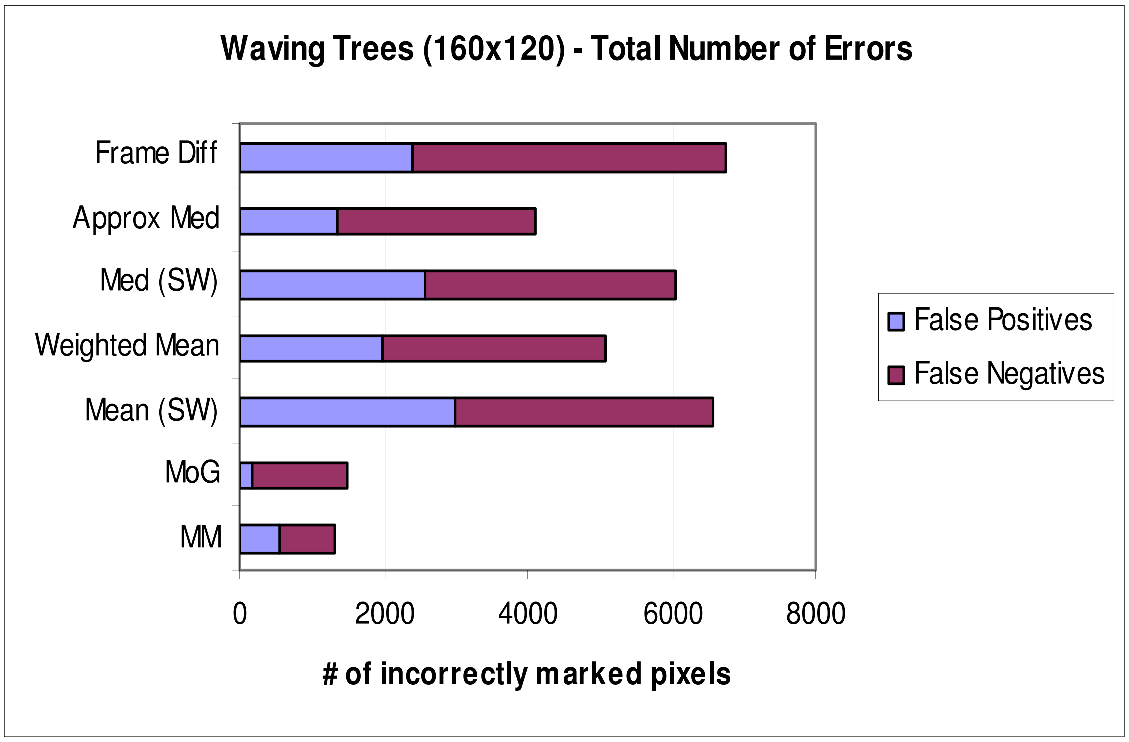


Figure 6 Waving tree errors

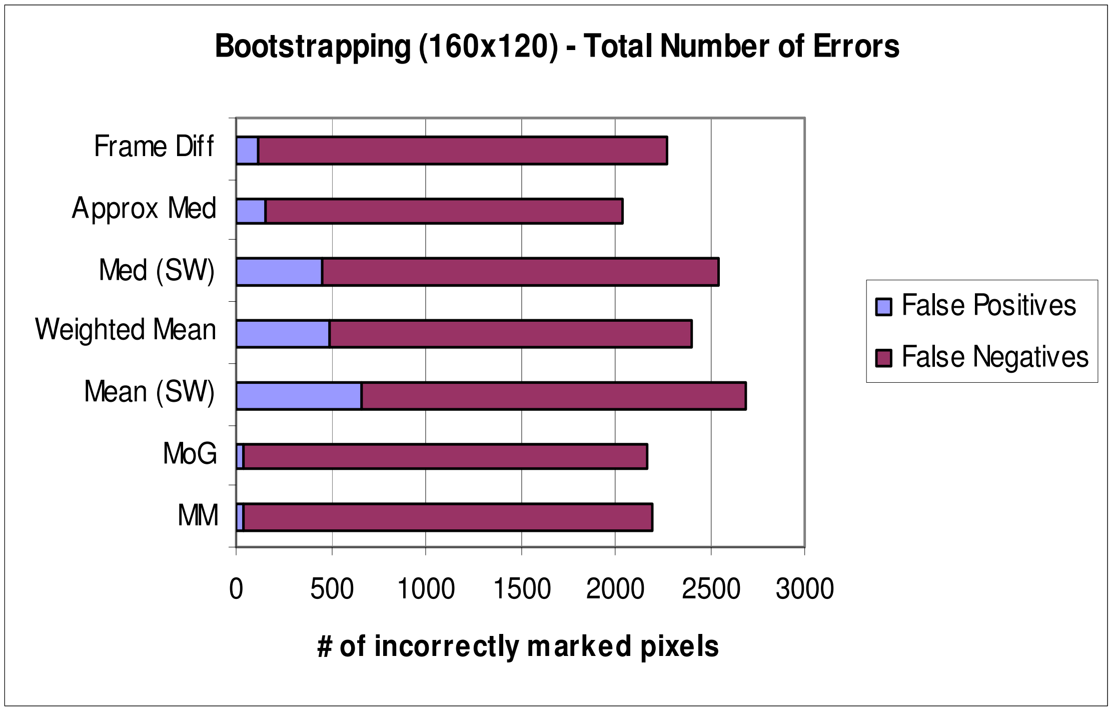


Figure 7 Bootstrapping errors

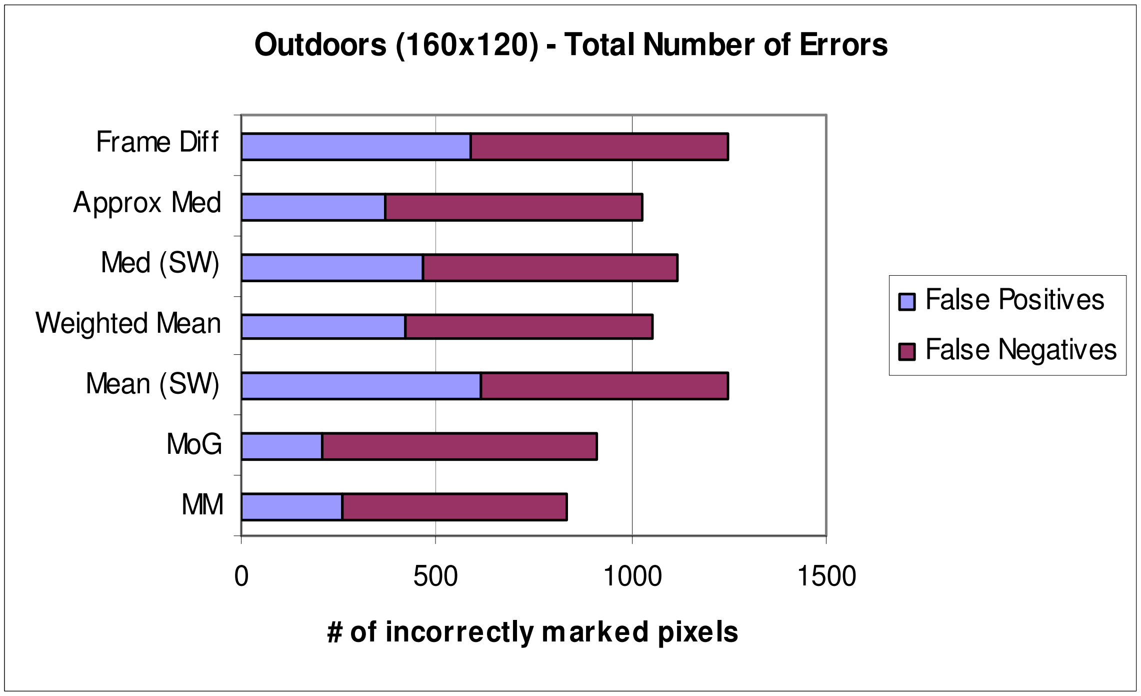


Figure 8 Outdoors errors

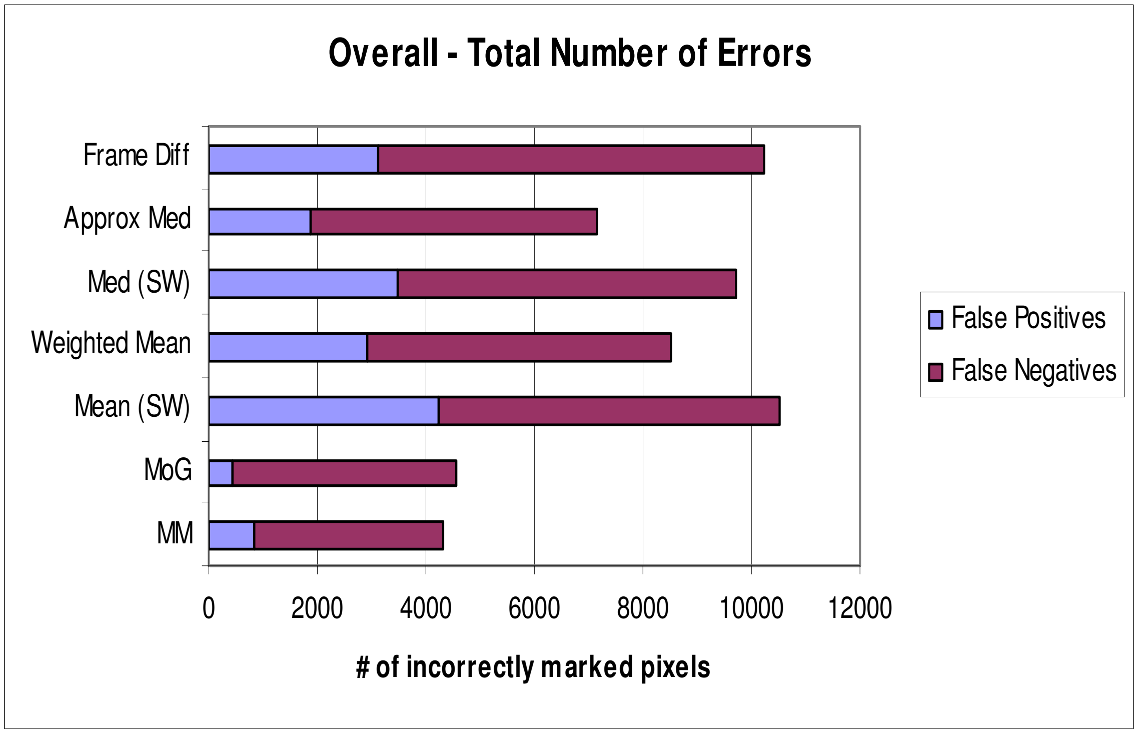


Figure 9 Overall errors

False positives indicate foreground identified 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, the MoG and MM techniques demonstrate comparable accuracy that is superior to the other methods.

Figure 10 displays the image quality for each background modeling 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.

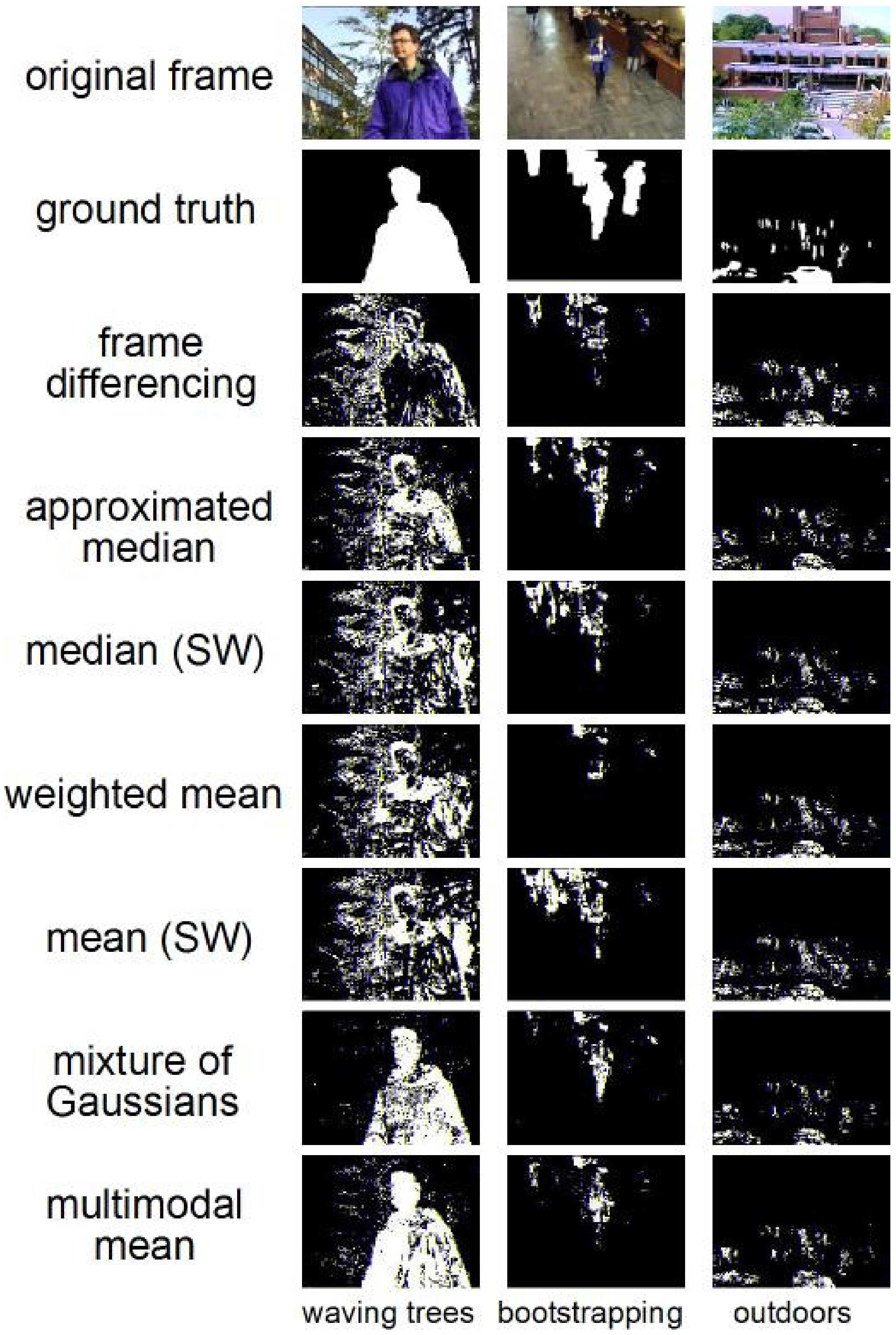


Figure 10 Image quality comparison of background modeling techniques

In Waving Trees, only the multimodal techniques incorporate the moving tree into the background. In Bootstrapping, all techniques are able to detect elements of the foreground identified in the ground truth. Unfortunately, the sliding window and weighted mean methods also identify reflected light on the floor (false positives). Outdoors features a large number of foreground elements as well as moving trees. Both multimodal techniques have significantly higher false positive accuracy.

Table 3 lists average processing times, average frame rates, and storage requirements for each method executing on the test platform. Because the sequence frames originated from standard files rather than camera output, I/O requirements are not included in these figures.

Table 3 Algorithm performance on test platform

Algorithm	**Time (ms)**	**Rate (fps)**	**Storage (words/pixel)**
Frame Differencing	7.6	32.0	1: packed RGB
Approximated Median	8.5	17.3	1: packed RGB
Median (SW)	69.2	4.4	3: 3 char × 4
Weighted Mean	26.8	7.3	1: packed RGB
Mean (SW)	28.2	5.5	3: 3 char × 4
MoG	273.6	.7	22: 5 FP × 4 modes + 2 int
Multimodal Mean	43.9	2.8	18: (4 int + 2 char) × 4 cells

The results showed that the MM method executes 6.2× faster than the MoG technique, while providing comparable image quality and accuracy. It also requires 18% less storage per pixel and uses only integer operations. Although many of the other methods offered lower execution times and storage requirements, their accuracy is insufficient for many applications.

3.6 Multimodal Mean on HP Platform

To further highlight the impact of compact representation and algorithmic complexity on embedded platforms the evaluation was repeated on a more capable execution platform. The HP Pavilion Slimline S3220N PC shown in Figure 11 was the chosen platform, and it featured full PC functionality in one third the conventional tower size. It measures at just over a foot long and less than a foot high. It has an AMD Athlon 64 X2Dual-Core processor with 512 KB cache and a 512 KB L2 cache. It also has an NVIDIA GeForce 6150 LE graphics processor, 1024 MB of DDR memory and a 250GB hard drive. The Slimline runs Microsoft Windows Vista as the operating system and Micrcosoft Visual Studio 2005 was used for application development. This platform has greater computational throughput, more main memory, and better floating point support than the eBox. This comparative analysis provides additional insight into algorithm demands and their performance on different embedded platforms.

Figure 11 HP Pavilion Slimline S3220N PC

For this experiment, two full frame (640 x 480) sequences were used to evaluate each background modeling technique because the Slimline, unlike the eBox, had enough memory to accommodate the full resolution test sequences. The first was the outdoor sequence used previously with a length of 901 frames. The second sequence was a 750 frame (640 x 480) outdoor walkway outlined by trees on a sunny day and was also taken with an inexpensive webcam. Under those real-world conditions, waving trees and shadows resulted in a dynamic background.

Table 4 lists average processing times per frame and average frame rates on the HP Pavilion Slimline test platform. The performance of MM on the HP platform was 4.23x faster than that of MoG, compared with a 6.2x improvement on the eBox.

Table 4 Algorithm performance on HP platform

Algorithm	**Time (ms)**	**Rate (fps)**
Frame Differencing	28.55	57.83
Approximated Median	34.29	48.16
Median (SW)	174.3	9.47
Weighted Mean	45.96	35.91
Mean (SW)	55.3	29.85
MoG	444.66	3.71
Multimodal Mean	105.07	15.71

While the improvement is partially due to less memory limitations and better hardware-supported floating-point computation capabilty, it is clear that reducing overall algorithm complexity and using a more compact data representation offers a significant performance improvement on higher performance embedded platforms. Therefore, faster and more capable hardware platforms alone are an insufficient solution to designing efficient embedded surveillance systems. The first half of this dissertation addressed this by providing a framework for redesigning software applications for efficient execution on embedded platforms.

CHAPTER 4

REAL-TIME ADAPTIVE BACKGROUND MODELING FOR MULTICORE EMBEDDED SYSTEMS

4.1 Introduction

Demand for efficient image processing on non-traditional platforms is being fueled by the proliferation of portable multimedia devices such as cell phones, gaming systems, media players, and automotive imaging systems. A popular solution among hardware vendors is scaling down versions of general-purpose processors and repackaging them as low-power embedded cores. Parameters like video frame rate and image resolution are scaled down to accommodate real-time performance. These techniques will not be sustainable as more complex applications are ported to embedded systems. Customized hardware, specially designed for embedded multimedia, will be required to meet the demands of this fast-growing market.

Current trends in microprocessor design integrate several autonomous processing cores onto the same die. Industry efforts, such as the Cell Broadband Engine from Sony, Toshiba, and IBM [33], Niagara from Sun [34], and Montecito from Intel [35], as well as university-led designs, such as MIT's RAW [36] and the University of Texas's Trips [37] are representative multicore architectures. Multicore architectures are particularly well-suited for image processing applications where it is typical to perform the same set of operations repeatedly over large datasets. However, there are still significant differences between general-purpose and image-processing workloads. Image-processing applications exhibit high levels of data parallelism and feature little data reuse.

Conventional general-purpose architectures are limited in this regard because they do not support the scaling of arithmetic units and registers to the very large numbers required for the concurrent execution of large groups of image pixels. Also, they have cached-based memory systems that are tuned for data reuse and hence are ill-suited for image processing[38]. In [45] an evaluation of a cache-based system and a direct DMA system is performed on the TI TMS320C6416 DSP [46] which provides support for both options.

The DMA-based system offers better performance than the cache-based one for embedded image-processing applications because it offers direct control of data transfers to the applications programmer and therefore ensures predictable access times. The results of the evaluation, shown in Table 5, demonstrate that for the performance-optimized PfeLib function PfeBayerLinearR, the DMA-based system had a 3x speedup over the cached-based system. In the IRAM configuration all frame buffers were located in internal memory and the L2 cache was not activated. The ERAM configuration was similar to the IRAM except that the frame buffers were located in external memory. In the L2CACHE configuration, 64KB of internal memory was configured as L2 cache, and the frame buffers were located in external memory. The cache was reset to clean before starting each run.

Table 5 Cache vs. DMA

Configuration	**Performance (cycles/pixel)**
IRAM	5.7
ERAM	860
L2CACHE	18.8

In [47], a similar evaluation is performed with a MAP 1000 processor for four applications - 2D convolution, affine warp, invert and add, and 2D fast Fourier Transform - and the results showed that the DMA-based system yielded better results.

Ideal candidate platforms for embedded computer vision must feature a large number of processing cores, a large register file, and hardware support for transferring large amounts of pixel data to and from processing cores. Also, they must meet embedded power, size, and cost constraints. Based on the criteria described the Cell Broadband Engine was identified as a suitable hardware platform for embedded early vision algorithms.

4.2 Cell Broadband Engine: An embedded multicore execution platform

The Cell B.E. (Figure 4) is a heterogeneous multicore chip which features one PowerPC (PPE) computing core and eight Synergistic Computing (SPE) cores on the same die. The PPE is a fully compliant 64-bit PowerPC RISC architecture with 32 128-bit vector registers, 32-KB L1 instruction and data caches, and a 512-KB unified L2 cache. It is a modified version of the general-purpose Power architecture and is tuned for executing general-purpose workloads. Each SPE is a 128-bit RISC processor with 128 128-bit registers and 256 KB of local storage. The SPEs are designed for high-performance, data-streaming, and data-intensive computation. DMA is the primary method of communication between the SPEs and main memory. 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. The entire system is well-suited for embedded image processing applications with the PPE handling program and data management and flow control, while the SPEs perform the pixel-level image operations[50].

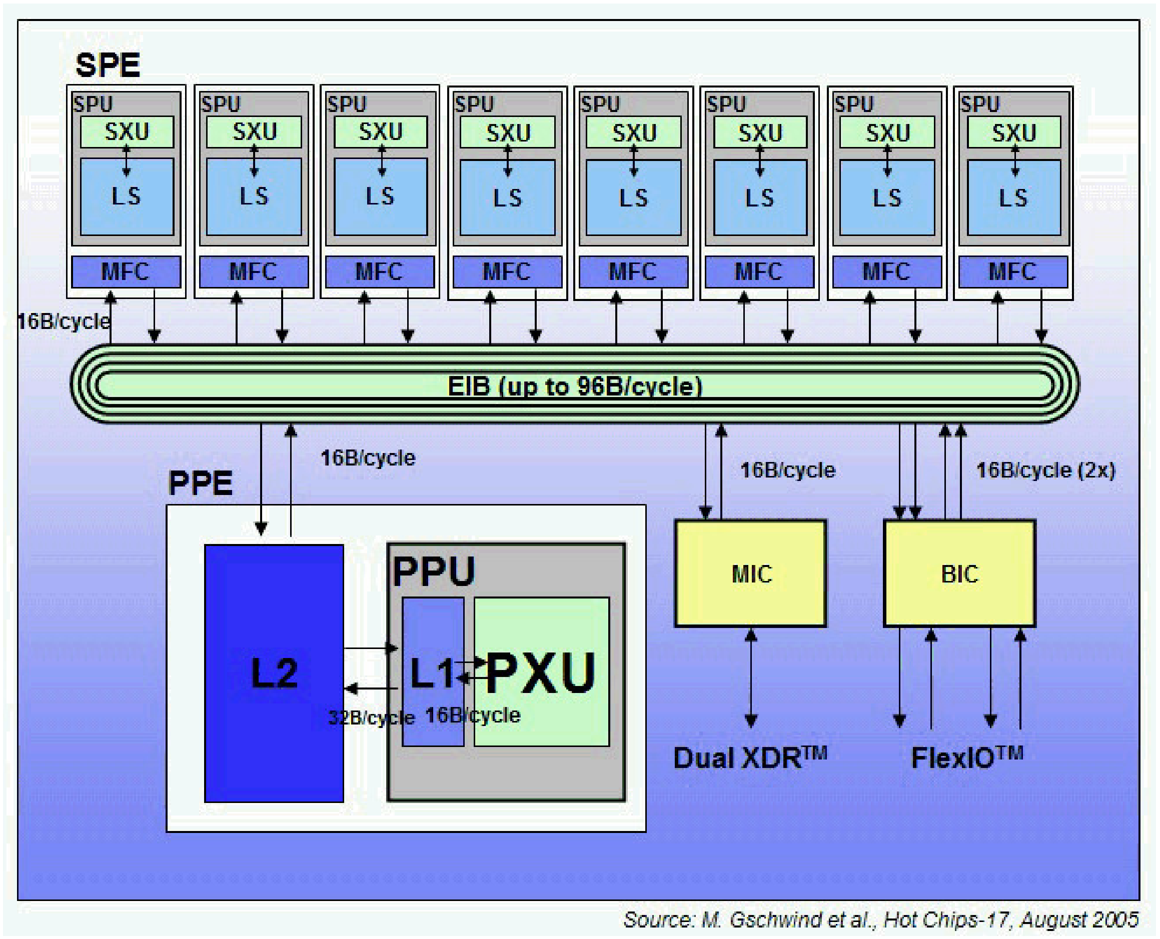


Figure 12 Cell architecture

4.3 Embedded Multicore Computer Vision

Multicore processor platforms provide tremendous potential to achieve real-time performance of computer vision applications. 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.

Early vision algorithms typically involve a small number of micro-operations performed over a large number of pixels. This makes them memory-intensive as well as computation-intensive. For example, a 720x640 pixel image in a standard RGB format requires 1.38 MB to store as a 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 early vision 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 460KB or a third the input image size.

For most systems, it will be nearly impossible to perform the entire background modeling 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 multicore embedded systems, however, limited memory decreases the processing block size and therefore more iterations are required.

Data domain parallelization [49], where 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* [49]*)* 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 [45].The next section shows the implementation of *Multimodal Mean* on the Cell B.E. platform and parallelization and processing optimizations that result in a 25% increase in performance over a baseline approach.

4.4 Baseline Processing of Multimodal Mean on the Cell Broadband Engine

The *Multimodal Mean* background modeling algorithm was implemented on the Cell B.E. and evaluated against the other background modeling techniques listed in Chapter 3. For this experiment, the test suite comprised of two longer outdoor sequences captured using an inexpensive webcam. Both sequences contained 700 frames and in each sequence, frame 453 was sampled for accuracy analysis. Also, the resolution was increased and both image sequences comprised of images with a resolution 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 recorded 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 conditions, 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. The various algorithm parameters were kept the same as those in Table 2.

The execution platform was a Sony Playstation 3 (Figure 13) with the Cell B.E. multicore processor running Yellow Dog Linux (YDL) 5.0.

Figure 13 Sony Playstation 3 with YDL

The background modeling algorithm was set up to leverage the strengths of the respective cores on the Cell. The general-purpose PPE was used for image decoding and encoding, core synchronization, and other book-keeping tasks. The SPEs, which are designed for high-performance, data-streaming and data-intensive computation, were used to perform the bulk of the background modeling algorithms. The SPEs are not cache-based and DMA is the primary method of transferring images between the computing cores and main memory. The maximum size of each DMA transfer is 16KB.

Although the evaluation is performed 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. Most processing cores should capably handle this dedicated task so there is no added benefit of having a general-purpose processor like the PPE on the Cell. Also, the SPEs 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 on-chip memory the advantages of the reduced storage features of the algorithm and the benefits of optimizing processing and partitioning techniques will be more pronounced.

The background modeling algorithms were implemented in C and compiled using gcc for the PPE and gcc-spu for the SPE. The background model was created and maintained by the PPE and different parts were transferred to each SPE along with the corresponding portion of the image to process. This arrangement was necessary because even the least memory intensive background modeling techniques (e.g. frame differencing) could not support the entire 720x640 image being processed in a single pass by all the computing cores. For the *Multimodal Mean* algorithm, the periodic decimation and recency resets were performed by the PPE. All other components of the algorithm were performed on the SPE. For all the other techniques, the entire algorithm was run on the SPE.

All images from the test sequences were in JPEG format and these were loaded onto the hard drive before each run. The independent JPEG library [39] was used to perform the image encoding and decoding.

4.5 Evaluation and Results

In this discussion, block size is the portion of a given image that is processed by a single SPE without a new iteration of data exchanges. To keep the processing balanced among cores the chosen block sizes were limited using the following constraint:

Image Size (pixels) *mod* (Block Size x Number of SPU) == 0.

For the first evaluation, the configuration that maximized block size was chosen. The SPE storage was divided into two areas; the first held a block of the current frame and the other held 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 6 shows the memory allocation on each SPU using this configuration.

Table 6: Memory allocation

Algorithm	Block Size (pixels)	Image Size (KB)	BG Model Size (KB)
Frame Differencing	38400	115.2	115.2
Approximated Median	38400	115.2	115.2
Median (SW)	12800	38.4	153.6
Weighted Mean	38400	115.2	115.2
Mean (SW)	12800	38.4	153.6
MoG	1600	4.8	160
Multimodal Mean	3200	9.6	153.6

The multimodal background models require storage for 4 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 storage and has half the block size.

Figure 14 and Figure 15 show the performance results obtained from running each algorithm on the Cell using the configuration described in Table 6. Because the sequence of frames originates from standard files rather than a camera output, I/O requirements are not included in these figures.

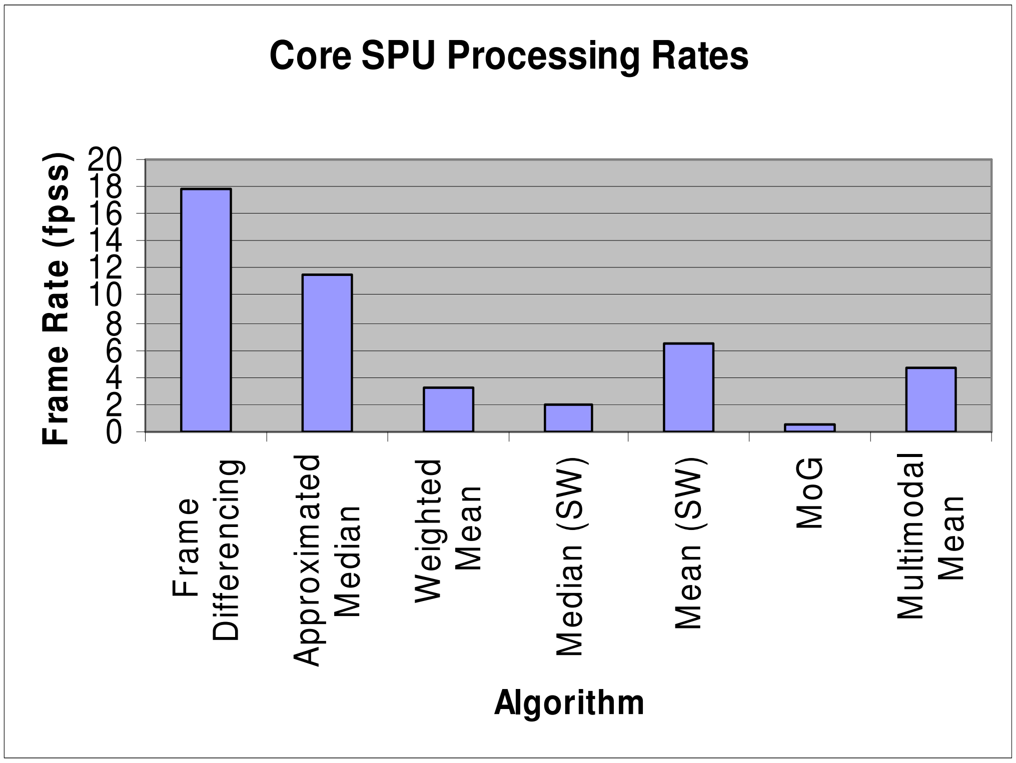


Figure 14 Algorithm performance in frames per spu seconds, excluding data transfer

The results in Figure 14 show 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 execution to the time the algorithm is completed. The spu_decrementer [53] was used to record the time spent executing the core algorithm on each SPE. Results are given in frames per spu seconds (fpss).

From the results it can be observed 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 9x 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 the uniprocessor eBox 2300 Vesa PC and the dual-core HP Slimline platforms described in Chapter 3. It also shows that even on a suitable platform such as the Cell B.E., algorithm design has a significant impact on overall system performance.

Figure 15 shows the overall algorithm performance in frames per second, including data transfer.

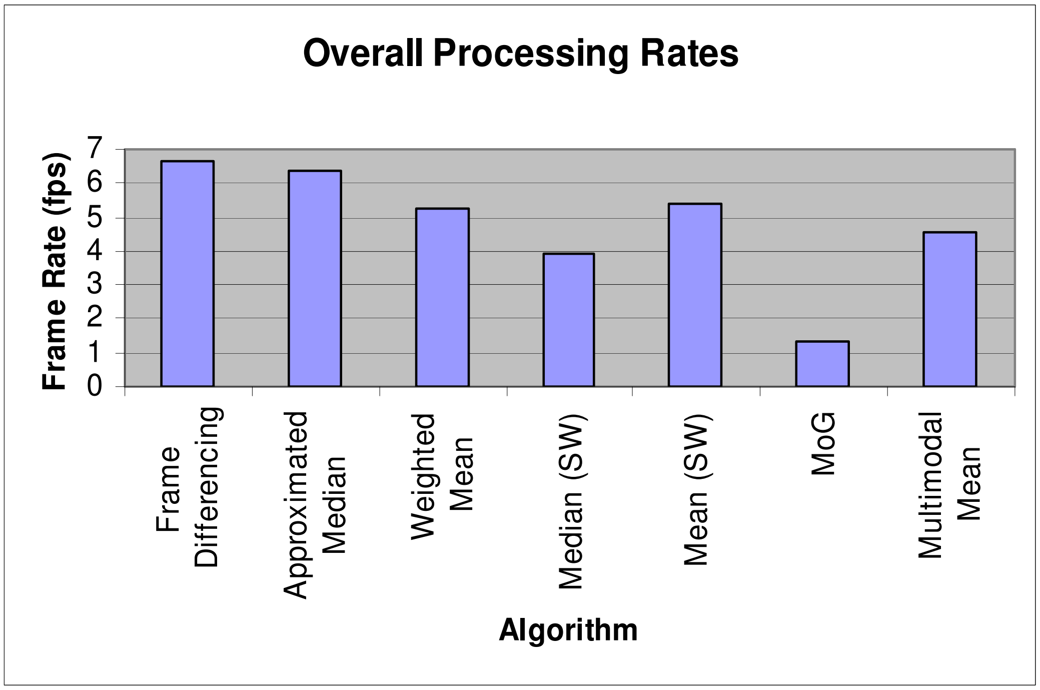


Figure 15 Algorithm performance in frames per second, including data transfer latency

Overall, the results show that MM achieves a 3.4x speedup over MoG and has comparable performance to the other techniques. In general, it is observed that the disparity between the performance of single-mode techniques and that of the multimodal ones, particularly MM, is narrowed. There are two reasons for the observed disparity. 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 16KB 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 4.8KB of image data during each iteration and this data transfer 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.

The disparities demonstrated between the core kernel execution time of the algorithm and the overall execution time (including data transfer) highlights the importance of optimizing the partitioning and processing of workloads for multicore processors. The performance gains of having several computing cores working concurrently are quite obvious, but the challenge is to keep the cores constantly working. The next section describes partitioning and processing techniques that optimize workloads for each core.

Figure 16, Figure 17, and Figure 18 quantitatively summarize accuracy for each technique. The same criteria for identifying false positives and false negatives used in the experiments in Chapter 3 were applied to this experiment. Generally, MoG and MM demonstrate comparable accuracy that is superior to the other methods as is shown in Figure 19. The accuracy results of the evaluation were consistent with those from the eBox and HP evaluations in Chapter 3.

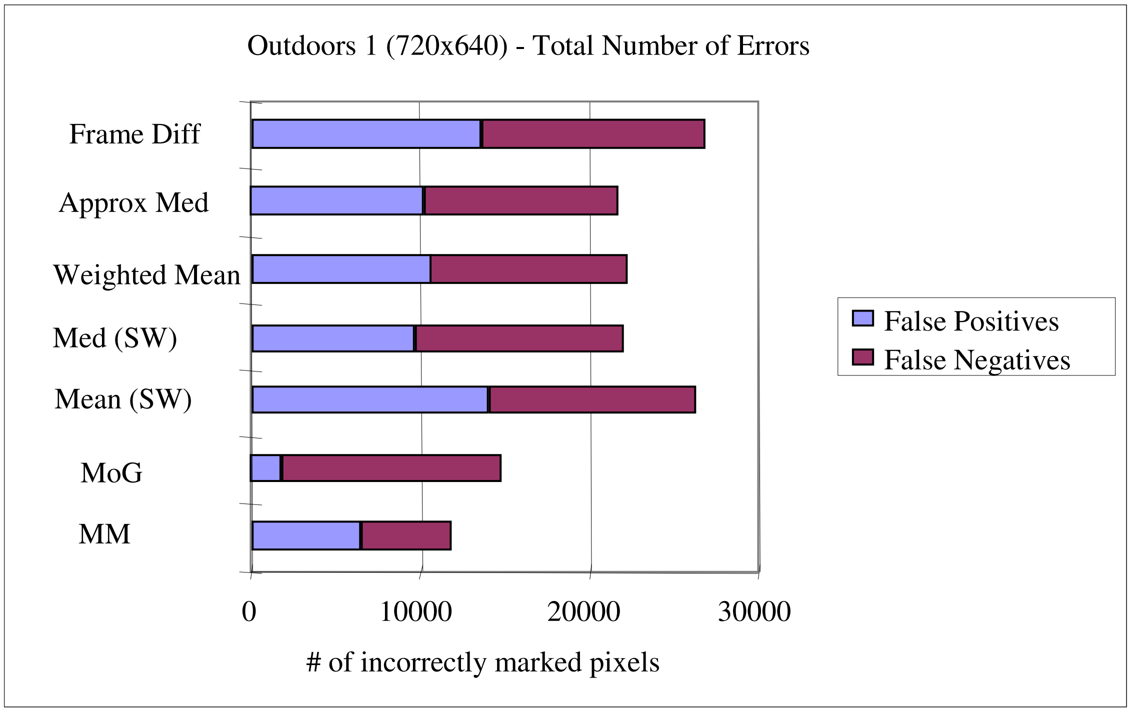


Figure 16 Outdoors I errors

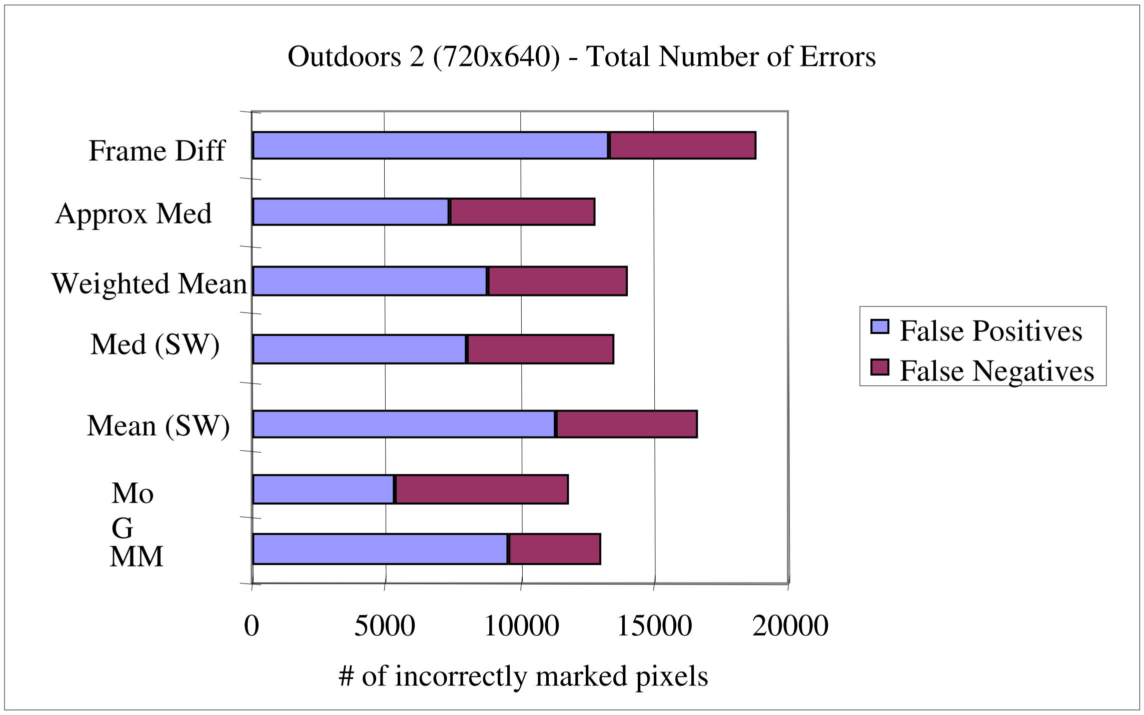


Figure 17 Outdoors II errors

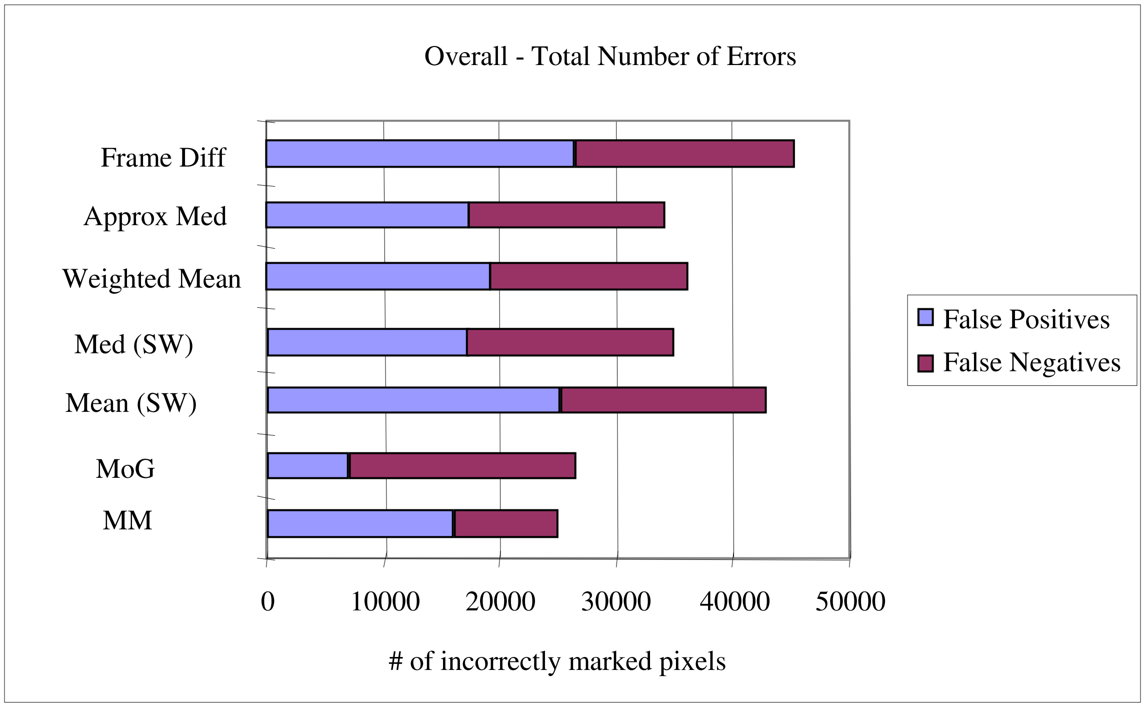


Figure 18 Overall errors

Figure 19 displays the image quality for each background modeling technique. Multimodal methods (MoG and MM) generally exhibit the lowest number of errors across the sequences and 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.

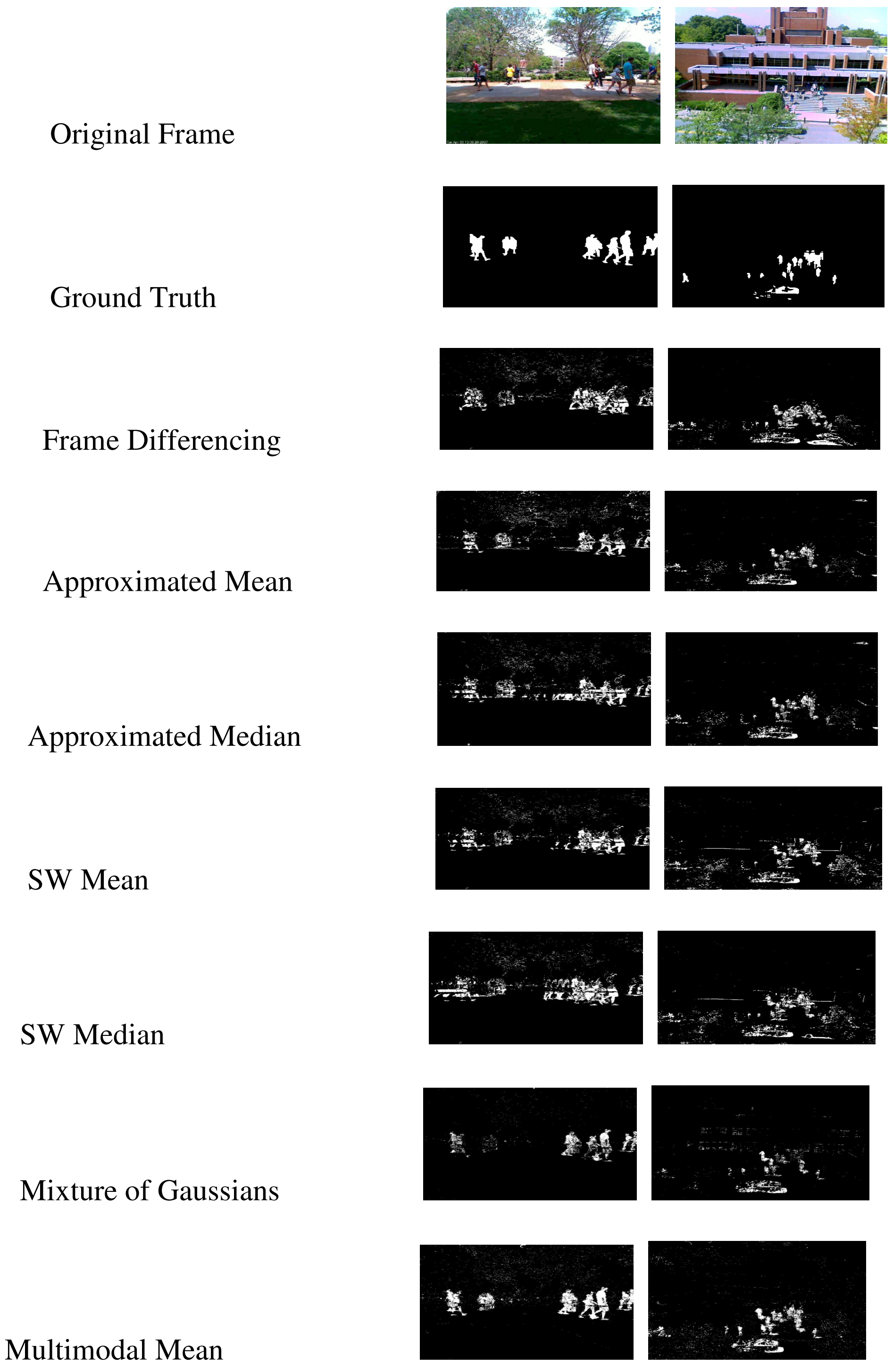


Figure 19 Image quality on Cell B.E.

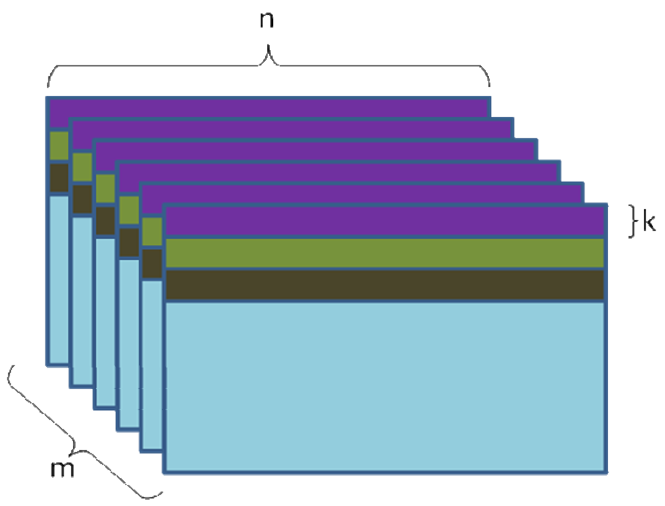
4.6 Tile Processing

Making data available to keep the computing cores busy on an embedded multicore processor is crucial to achieving high performance and efficiency when executing embedded early vision algorithms. This section introduces a tile processing workload partitioning arrangement and shows the impact on the overall processing.

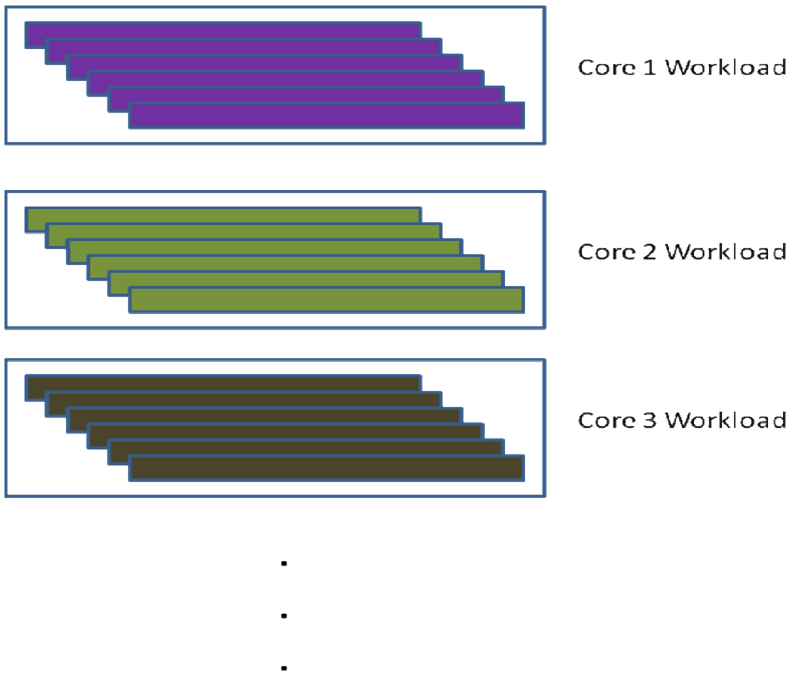
Typically, live video input from the webcam is buffered as images by the camera driver. Rather than divide the workload in the current frame using maximum block size and available memory on the SPUs, a tiled workload was created from the buffer using much smaller block sizes called *tiles*. Tile sizes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

Image Size *mod* (Tile Size x Number of SPU) = 0,

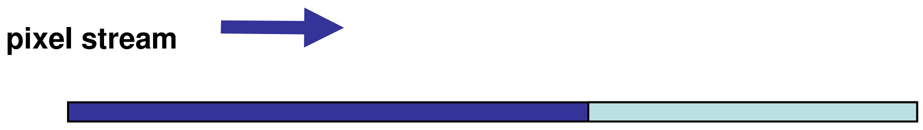
where Image and Tile Sizes are measured in pixels.



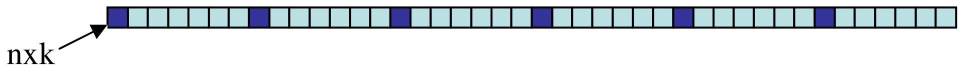
a. Full images



b. n x k tiled workloads processed by each core



c. Regular buffering of pixel stream as contiguous block



d. Buffering pixel stream in groups of n x k pixels

Figure 20 Tiling

In this discussion a buffer containing *m* of these images is considered. An *n x k* tile is selected from the same location for each image in the buffer as shown in Figure 20a. *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 Figure 20b.

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 20c 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 20d shows the stream broken up into groups of *n x k* pixels. The first group of *n x k* pixels in the first image is stored at the beginning of the storage array, after which *(m-1) x n* x k pixel locations are skipped in the array before storing the next group. This process is repeated for the entire first image stream. For subsequent images, the first group of *n x k* pixels in an image *j*, begins at pixel location *((j-1) x n x 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 the effect of using this technique was evaluated on the Cell B.E. platform. The tiled buffering method was benchmarked 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 image. Each scanline was stored using both techniques described above and the total time taken to decode and buffer each frame was evaluated. For the evaluation, the resolution of the images was 720 x 640 and the average was taken from decoding 100 images. The results showed that for different buffer sizes the tile buffering does not significantly increase the image decode times. On average, a 0.023% increase in decode time per frame was observed when using tile buffering. Also, there is no notable increase in total time as the buffer size was increased. Furthermore, the image retrieval times are significantly less than the actual background modeling 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. This process is analogous to caching the entire background model and using the cached model to process successive images in a video sequence while updating the background model. After processing a given series, the background model on each core is written back to main memory in a process similar to the operation of a write-back cache. Because different portions of the background are processed by each core there are no coherency issues and the associated complexities of a shared-memory model are avoided. Table 7 shows the storage requirements for each configuration. All configurations use at most 230KB of the SPU storage to allow for run-time memory requirements.

Table 7: Image storage requirements

Buffer Size	Image Size (KB)		
	Tile =1600 BGM=76.8KB	Tile =2400 BGM=115.2KB	Tile =3200 BGM=153.6KB
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 8 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 1600 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.

Table 8: Performance

Buffer Size	Frame Rate (fps)		
	Tile =1600 BGM=76.8KB	**Tile =2400 BGM=115.2KB**	**Tile =3200 BGM=153.6KB**
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		

Using the tile buffering technique will cause a multicore, video-surveillance system to incur a slight buffering latency due to the temporal buffering. As a consequence, the reaction time of the system could slightly increase. 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 at the front-end of the system (camera driver). Rather, it is the transfer of data to execution units and the computation of the kernel of the early vision algorithm and improvements to this component will significantly impact overall perfromance.

For example, the Playstation Eye USB Camera for the Playstation/Cell B.E. platforms buffers full-resolution 640x480 images at 60 fps. Using the tiling technique with buffer size 32 will result in a delayed system reaction time of 500ms 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 320x240 images will result in a delay of only 125ms.

However, the benefits of the tiling technique are evident with the increased processing bandwidth. For example, an end-to-end application running at 32 fps will be able to run at 24 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.

CHAPTER 5

CAT-TAIL DMA: EFFICIENT IMAGE DATA TRANSPORT FOR MULTICORE EMBEDDED SYSTEMS

5.1 Introduction

The previous chapter highlighted the importance of efficient data management to the overall performance of multicore embedded vision systems. It showed how temporal buffering of images combined with a shared background model reduces the number of data transfers required for video surveillance algorithms. This data management technique helps improve efficiency by reducing the number of data transfers but does not address the actual transfer mechanism. Techniques that reduce the latency of the actual data transfers could further improve the performance of such systems.

Typically, image-processing applications require the transfer of large amounts of data between the execution units, where the images are processed and off-chip memory, where the images are stored. The high-throughput and low-latency characteristics of these applications make image transport crucial to the overall program performance especially on multicore systems where several cores need to be furnished with data.

To fully leverage the concurrent execution of several powerful cores in imaging applications, a very fast, high-bandwidth communication network is typically provided to move data throughout the system. Large data transfers are performed over this network through DMA and each processing core has a DMA-controller to which it can offload block data transfers to memory as well as to other processing cores. Multiple cores can potentially generate several DMA transfer requests and regular cache block requests to memory at once. Arbitration is performed in hardware through the bus arbiter to determine which core gets access to the bus during contention. In addition, the arbiter determines when source and destination transfer paths don't overlap in which case multiple transfers can be performed concurrently on the bus.

Performing data transfers through direct DMA access places a greater burden on the programmer to make the best use of the resources provided by multicore architectures. Partitioning of the multimedia workload to most efficiently utilize all the cores is critical to overall program performance. In addition, data transfers to and from the cores are now in the programmer's domain and efficient management of DMA transfers along with program execution have a direct impact on execution times. In the presence of multiple cores, these DMA transfers have to be carefully managed to prevent collisions which may result in data transfer bottlenecks.

The potential to incur higher data-transfer latency due to collisions is magnified on multicore embedded systems because of the limited amounts of local storage available on the execution cores. For applications like image and video processing, which involve very large datasets, limited local storage means a small portion of a given frame can be operated upon in a single iteration on each core. Several iterations may be necessary to process the entire frame but concurrent execution on the individual cores can collectively yield tremendous speedup. However, processing smaller blocks of an image also increases the frequency of data transfers between main memory and the local storage on each core. Furthermore, the presence of several cores requesting data from main memory simultaneously increases the potential for DMA collisions and can result in higher data-transfer latency.

This chapter evaluates existing DMA-buffering techniques and identifies the challenges faced when employing them to execute early vision algorithms on multicore embedded systems. It presents a new technique called *cat-tail DMA,* which addresses some of the shortcomings the previous techniques, and provides low-overhead, globally-ordered, non-blocking DMA transfers on a multicore system. With this method the data-transfers times are reduced by 32.8% for the *Multimodal Mean* background modeling algorithm while increasing the utilization of core local storage by 60% over existing double-buffered techniques.

5.2 Background Work

Parallel image processing on multiprocessor systems was the focus of significant research even before the advent of multicore processors. Ni et al. [53] present a multiprocessor system for image processing of office documents and evaluate the scheduling policies of the processors. They constructed a prototype system with one master processor and four slave processors connected by a shared bus. They considered two processor scheduling techniques under this model. The three-step overlapping policy separately treats the storing of the image segment currently being processed and the loading of the next image segment to be processed. The two-step overlapping policy loads the next image segment to be processed immediately after finishing the storing of the previous image segment.

They concluded that to optimize the performance of the system,

- the processor scheduler should be made as simple as possible;

- the scheduling overhead should be relatively small compared to the image segment transfer time. In other words, the image segment size should be made as large as possible;
- if the image processing time is very large compared to the scheduling overhead, the scheduling overhead will become negligible.

Their conclusions are even more applicable to multicore processors where the communication latency is much less because it involves chips on the same die. Also, with much higher processor speeds a complex and slow scheduling process will quickly and easily lead to data-transfer bottlenecks.

Lee et al. [54] propose a compile-time processor assignment and data partitioning scheme that optimizes the average run-time performance of task chains with nested loops. They developed a library of computer vision and image processing operations and built a model to classify data-dependent and data-independent operations and to tabulate the costs of many popular pixel and masking operations. Also, they modeled data redistribution costs through both all-to-all communication primitives as well as between any two data distribution schemes.

They ran an algorithm at compile time that uses information from the specified tasks to determine suitable processor assignment and image data-partitioning schemes and generated parallel codes by employing existing parallel routines such as ScaLAPACK [55]. They evaluated the partitioning and scheduling schemes and parallel versions of several CVIP algorithms on MEIKO CS-2, a distributed memory parallel machine with a fat-tree-based communication network and a SUN SPARC Viking processor at each node. Their results showed up to a 50% speedup over unscheduled code.

By performing all optimization at compile-time this approach avoids any scheduling overhead which could increase program execution time. However, the drawback to this approach is that the method failed to dynamically capture workload variations during actual program execution.

Zhang et al. [57] presented a study on adaptive workload assignment while performing the Motion Picture Expert Group 2 (MPEG2) video encoding algorithm on a multiprocessor system. They chose the MPEG2 application because it compresses video data by macro blocks (MB) and the processing of each MB is fully independent. Since a frame of video usually contains a large number of MBs, this application was well suited for fine-grained partitioning.

Because of the non-stationary nature of most video sequences, motion activities are not distributed uniformly over a frame. The authors ran simulations that showed that the computational costs of certain MBs in areas with greater motion activity were about 3x those in other areas with little or no motion in the same frame. They measured the cost of processing a load from an encoded frame. Using this information they estimated computational load distributions that optimized performance among the processors for the next frame in the encoding order of the same picture type. This scheme was simple to implement and therefore resulted in very little overhead. Also, the technique allowed the program to adapt to changes in data and was not fixed at compile time.

They evaluated the scheme with three video sequences, *Football, Claire*, and an industrial experiment *EFE,* that were typical in terms of the motion activities in their respective picture scenes. The *Football* sequence had large fast-motion activities that were distributed globally across the whole picture scene. In contrast, the *Claire* sequence was local in nature and only had small slow-motion activities that were located in a small part (e.g., face) of the whole picture scene. The degree of motion activities in the *EFE* sequence was moderate, between the *Football* and *Claire* sequences.

The simulation was run on a single INMOS T805 processor used to simulate a multiprocessor system of N processors. The values of N used were 2, 4, 8, 16, and 32, and each video sequence had 100 frames. Their results showed up to 20 % improvement when using adaptive workload distribution over uniform distribution.

The approach above provided a dynamic workload management scheme but presents some challenges in a real multicore system. The hosting core will have to wait for communication from the other computing cores before making a decision about the next computational load. This could result in a data transfer bottleneck since several cores might request new data at the end of their computations. Also, this scheme cannot be extended to other applications that are not segmented into uniform macro blocks.

The discussion above shows highlights some of the data-transfer and scheduling challenges faced when executing early vision applications on a multi-processor system. For next-generation multicore systems these issues need to be address to achieve efficiency and high performance.

5.3 Single vs. Double Buffered DMA

Even with the advantages of using DMA for data transfers when executing image –processing applications, several techniques have been proposed to hide the memory latency of the data transfer transactions. In [48] a software prefetch mechanism is combined with DMA to hide memory latency on multimedia applications. In [45] optimum resource slicing is performed with double buffering for more efficient embedded image processing. These techniques apply to uniprocessor systems and DMA-based image processing on a multicore system offers unique challenges because of the introduction of multiple computing cores and multiple DMA-controllers.

The 2D block transfer mode is the most popular DMA method used in image processing. With this method, an image is split into blocks which are transferred from main memory to the processor and returned after processing. The block size is determined by the maximum allowable DMA transfer per transaction and the particular operations being performed on the image. Overlapping portions of a given block is common in some applications (e.g. edge detection) to compensate for the artificial boundaries introduced into the image during block segmentation. Extra processing between blocks is required in some more extreme cases.

Figure **21** illustrates the execution of an image processing application on a multicore processor using the single-buffered DMA mode. Block *n* of the input image, which is located in main memory off-chip, is transferred using DMA to buf_0 of processing core *n* for processing. A new DMA transaction is initiated in each core at the end of the processing for each block, and the processing core waits while the DMA-controller writes out the processed image and reads in a new one. The processing core does no useful work while the DMA-controller is performing the transfer.

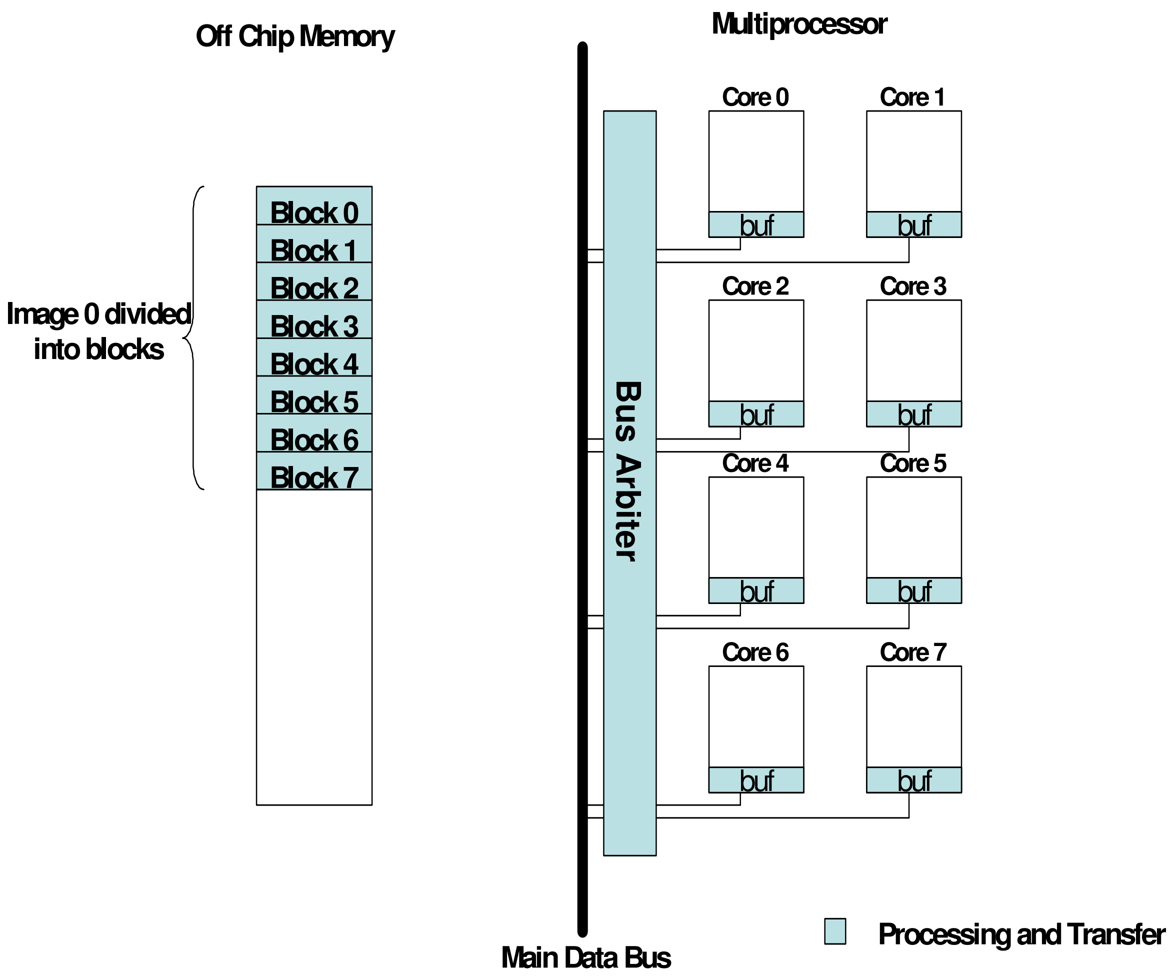


Figure 21 Single-buffered DMA

The double-buffering technique hides the data transfer latency by continuing execution on the processor while the DMA transfer is being handled by the DMA-controller. Figure 22 illustrates how the image processing application is executed in double-buffering mode. With this method, the memory transfer latency that occurs when a processing core is stalled for an old image to be written and a new one to be read is hidden by overlapping the execution of a given block with the transfer of the next.

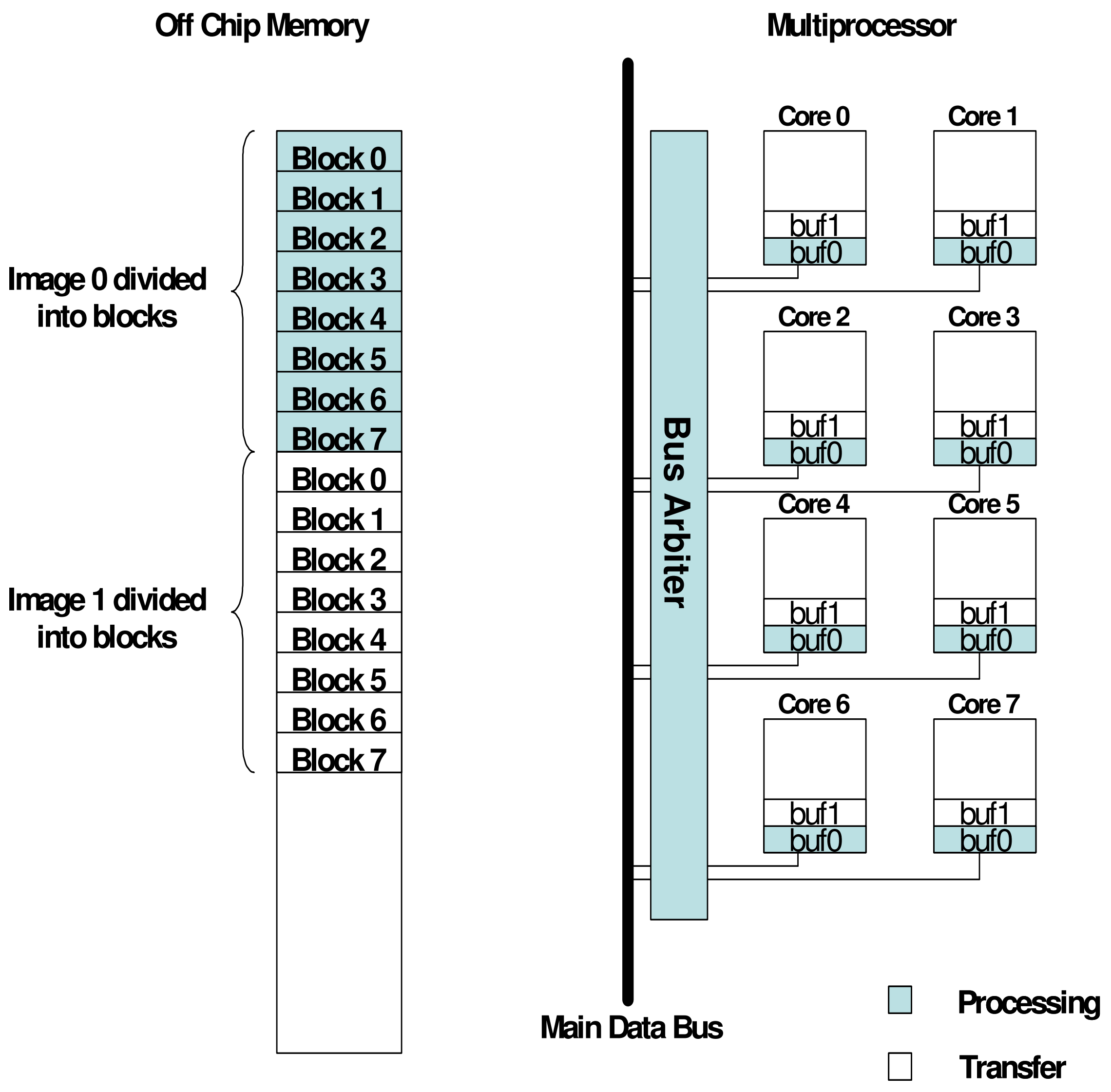


Figure 22 Double-buffered DMA

Two buffers, each of which can store a block of the image, are allocated on each core. During the processing of block *n* of the current image in buffer *j*, a DMA transfer is initiated to concurrently write out the processed data in buffer *j+1* and fill it with block *n* of the next image. Typically, the total time to process the entire block is greater than the latency of the DMA transfer in both directions, therefore a new block is available in buffer *j+1* by the time the processing of the old block in buffer *j* is complete. The steps to perform the double-buffered dma are as follows:

```
Read image into buffer j
Write tag 0
For i in 1->loopend:
    Poll status of tag (i&1)
    Read image into buffer j + (i&1)
    Write tag (i&1)
    Poll status of tag ((i-1)&1)
    Process buffer j+((i-1)&1)
    Write image from buffer j + ((i-1)&1)
    Write tag ((i-1)&1)
```

A tag is associated with each batch of data transfers and polling the status of a transfer channel for a given tag informs the processor whether that channel has completed the transfer. In the main loop, the processor checks for completion of the previous write transaction to the transfer buffer, schedules a new read transaction to that location, processes the data in the processing buffer, and schedules a write out of the processing buffer. Processing the data between the read and write transactions ensures that the likelihood of the read transaction being completed before scheduling the write is very high. This results in minimal waiting, if any. Similarly there is some loop overhead between the last write and the next read in an iteration which again minimizes the wait time.

For early vision algorithms such as background modeling, double-buffered dma presents several challenges for multicore embedded systems. Typically, they involve comparing the current image to a reference model, both of which have to be present on the cores for processing iteration. As a result, these algorithms require the transfer of multiple sets of data for each processing iteration, and require explicit management by the processor to ensure correctness.

Table 9 shows the block configurations partitioned to store the image and background model that maximized the local storage utilization on the Cell SPE. Frame differencing, approximated median, and weighted mean share the same background modeling data structure and are represented by a single mode. Similarly, both sliding window techniques are represented by sliding window. The multimodal algorithms were limited to 4 modes and the maximum-size of DMA transfers on the Cell B.E. is 16KB. Each SPE local storage area is 256KB and this is divided into two equal-sized image storage areas and two equal-sized background storage areas for double-buffered DMA.

Table 9 SPU Maximum block transfer for double-buffered DMA

Algorithm	**Block Size (Pixels)**	**Image Size (Bytes)**	**BG Model Size (KB)**
Single Mode	19200	57600	57600
Sliding Window	6400	19200	76800
MoG	800	2400	80000
Multimodal Mean	1600	4800	76800

From the table it can be inferred that to process a given block for a technique like MOG, a single DMA transaction is needed to transfer the image but 5 transactions are required to transfer the background model. Also, the entire background model must be available on the SPE to process the block. Similarly, the frame differencing technique requires 4 image transfer transactions and 4 background model transfer transactions.

The requirement for multiple data transfers of different datasets to complete one block of processing for computer vision applications is vastly different from traditional double-buffered DMA where a single read/write transactions pair is overlapped with processing. This inherently serializes the data transfer transactions because the processor must verify the completion of all block write transactions before scheduling block reads since both operations share the same buffer. Also, using double-buffered DMA decreases the maximum block size that can be processed because the local storage area must be split. This increases the frequency of block transfers and thus increases the probability for collisions which can lead to high data transfer latency for entire block transfers.

Furthermore, this problem can be exacerbated as the number of cores is increased on multicore systems where those cores are executing the same program and there is a high potential for several simultaneous DMA requests from competing cores.

5.4 Cat-Tail DMA

Cat-tail DMA addresses the issues discussed in section 5.3 by providing a technique for low-overhead, globally-ordered DMA transfers among processing cores that minimize collisions and reduce data transfer latency. This is achieved through:

1. Dividing the processing of blocks into two phases: the processing/transfer scheduling phase and the processing phase.
2. Staggering the execution on the computing cores to ensure that there is a contention free period to schedule transfers for each core.

The main concept with this technique is to reserve a unique period on each core where a series of large data transfers can be performed by the DMA controller with minimal input from the microprocessor.

5.4.1 Core Processing

In this discussion the transfer of two datasets *dataset A* and *dataset B* is considered. Two circular buffers, one for each dataset, are maintained in the local store area of each core. Unlike double-buffered DMA, the size of the circular buffers is constrained by the available local storage on the processing core and not the maximum DMA transfer block size.

The circular buffer is divided into two dynamic regions called the *transfer region* and the *processing region* as shown in Figure X. The processing is also divided into two phases. In the first phase the *transfer/processing mode* is used where a given portion of the *processing region* is processed while the data in the *transfer region* is written out and new data read in. The second phase uses the *processing mode* where the remainder of the *processing region* is processed and there is no data exchange. Two pointers are maintained for the circular buffers: a *processing pointer* points to the beginning of the next *processing region* and a *transfer pointer* points to the beginning of the next *transfer region*.

Figure X shows the progression of both pointers along the circular buffer in each core and the process is the same for both image and background. In this discussion a pointer is moved to the beginning of the buffer once it reaches the end (circular buffer).

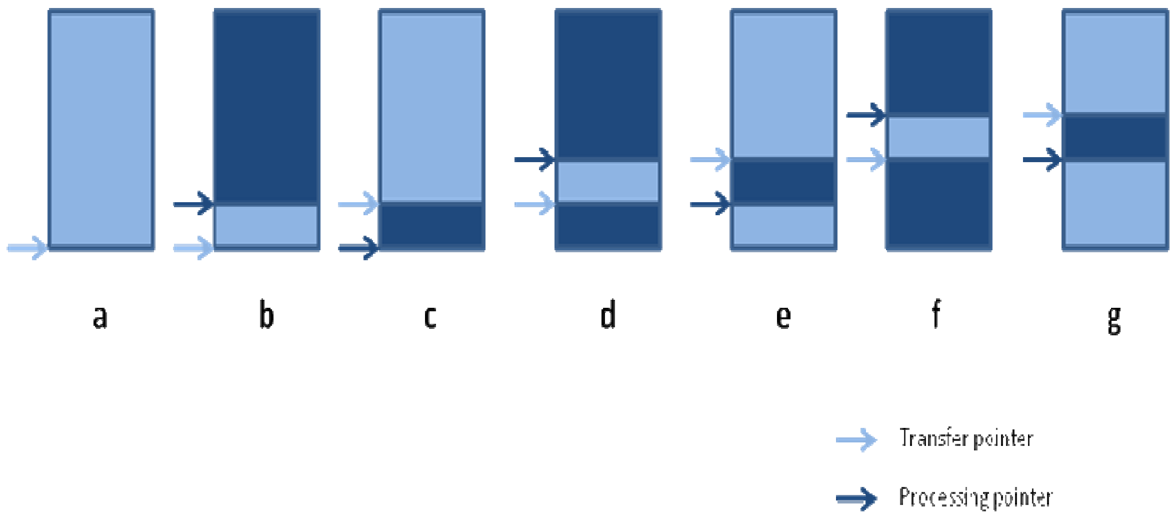


Figure 23 Circular buffering

At initialization, the entire circular buffer is filled with data and the *transfer pointer* updated to the end of the buffer (beginning of next *transfer region*) as shown in Figure 23a. The entire *processing region* is then processed in *processing mode* (no data exchange) and the *processing pointer* is moved to the beginning of the next *processing region* to complete the initialization as shown in Figure 23b.

After initialization the two-phase processing is used. Figure 23c shows the first phase processing using *transfer/processing mode*. A portion of the processing region is processed concurrently with the exchange of data in the *transfer region*. The *transfer pointer* is then updated to the beginning of the next *transfer region* and the *processing pointer* is updated to the beginning of the second phase of the *processing region*. The second phase of processing using *processing mode* is shown in Figure 23d. The remainder of the *processing region* is processed and the *processing pointer* is updated. However, there is no data exchange during this phase and the *transfer pointer* remains unchanged. This process is repeated in each core until the end of the program.

5.4.2 Staggered Execution

Execution on the cores is staggered to provide individual cores with a unique time slot to schedule data transfers in the first phase of processing described in Section 5.4.1. Figure 24 illustrates the staggered execution on the multicore system.

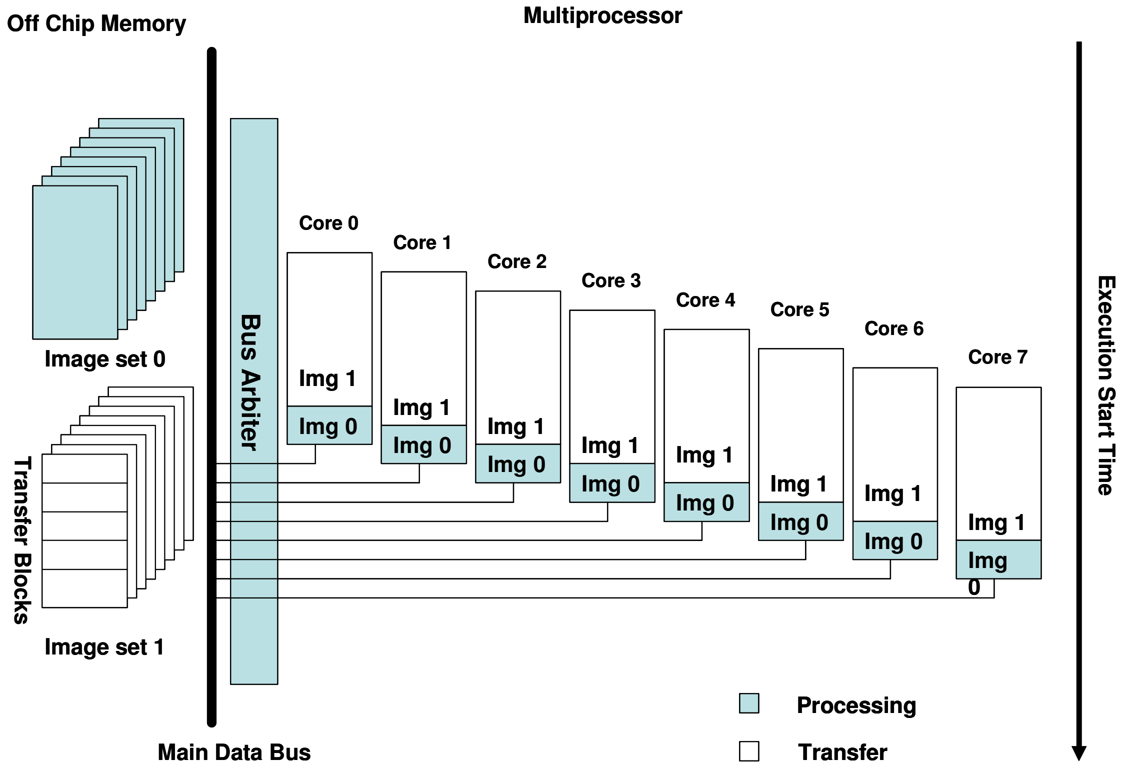


Figure 24 Staggered execution

A token is passed around cores in a round robin fashion to signal which core is scheduled for data-transfer. During the first phase processing, the processor on the core that possesses the token schedules all its data writes (*dataset A* and *dataset B*) interleaved with the processing of the image. The processor is not stalled to wait for completion of the transfer and the processing is continued. On the *kth* iteration of processing in the phase, the processor checks for completion of the scheduled writes and proceeds to schedule reads. On the *2kth* iteration the token is released to the next core and the process is repeated.

The execution is summed up as follows:

Phase 1:

```
Initialize process_counter
For i in 0 -> blocks in dataset A transfer region:
    Schedule write dataset A block
    Write datasetA_write_tag
    Process data in processing region
   process_counter++
For i in 0 -> blocks in dataset B transfer region:
    Schedule write dataset B block
    Write datasetB_write_tag
    Process data in processing region
   process_counter++
For i in process_counter->k
   Process data in processing region
Poll status of datasetA_write_tag && datasetB_write_tag
For i in 0 -> blocks in dataset A transfer region:
    Schedule read dataset A block
    Write datasetA_read_tag
    Process data in processing region
For i in 0 -> blocks in datasetB  transfer region:
    Schedule read dataset B block
    Write datasetB_read_tag
```

```
            Process image in processing region
        For i in k + process_counter->2*k
            Process data in processing region
        Release token
        For i in 2*k-> processing region
            Process data in processing region
    Phase 2:
        For i in  processing region
            Process data in processing region
```

Unique *dataset A* read, *dataset B* read, *dataset A* write, and *dataset B* write tags are maintained for data transfers as a mechanism to verify that transfer of data to a particular region is complete before attempting to process it. Also barrier options are used with the DMA transfer to ensure proper ordering.

5.5 Evaluation and Results

An experiment was designed to evaluate the performance of cat-tail buffering on the The Playstation 3 featuring the Cell B.E. For this experiment, the data structures for the background modeling algorithms described in the previous sections, as well as the associated portion of the image to process constituted the datasets. The entire background model was initialized by the PPE and held in main memory, and during processing selected portions were transferred as a dataset to each core. Similarly, images were decoded by the PPE and stored in a buffer in main memory at startup and during processing selected portions were transferred to each core. The resolution of the images was 640 x 480 pixels. The single buffering technique was used as the baseline. Both buffering techniques were implemented in C and the data transfers were evaluated for the data structures described. Table 10 shows the total size of blocks (in pixels) held on each core using cat-tail buffering. It shows that the utilization of local storage is improved by 60% when compared to ping-pong buffering (Table 9) because the separate buffers reserved to transfer and process images are not of equal size. Also shown is the *transfer region* which is the portion of the block that is exchanged in a data transfer as described above.

Table 10 Block sizes

Algorithm	**Blocks Held (Pixels)**	**Blocks Transferred (Pixels)**
Single Mode	38400	30720
Sliding Window	12800	10240
MoG	1600	1280
Multimodal Mean	3200	2560

All data transfers were 128-byte block aligned for transfer on the Cell. This influenced the maximum size of blocks held on each SPE core. For the single-buffering technique the same sized blocks were used for data transfers.

Mailboxes were used to communicate between SPEs and the PPU and signals were used for inter-SPU communication [50]. This allowed DMA transfers of data to be performed with minimum intrusion and very little communication overhead.

Figure 25 and Table 11 show the performance of both techniques the single and cat-tail buffering techniques.

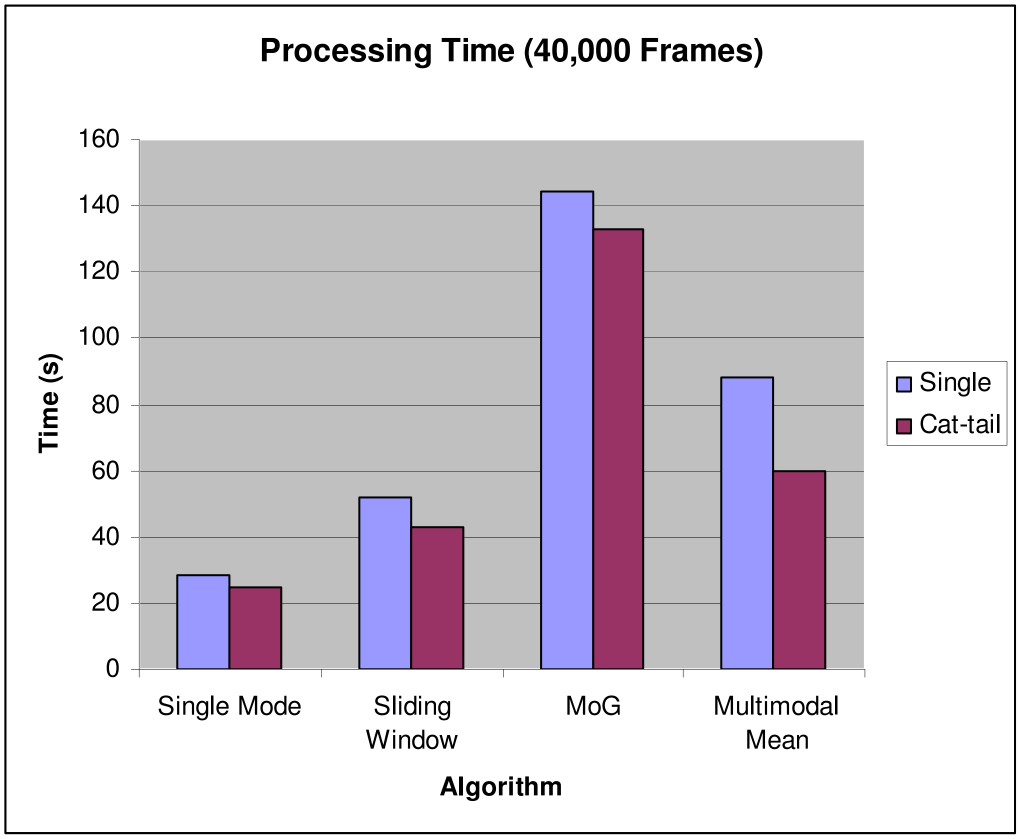


Figure 25 Performance of buffering techniques

Table 11 Buffering execution times

	Single	**Cat-tail**
Single Mode	28.11s	25s
Sliding Window	51.67s	42.94s
MoG	144.09s	132.55s
Multimodal Mean	88.19s	59.53s

From the results the *cat-tail DMA* technique showed a 32.8% reduction in total data transfer time over the baseline for multimodal mean. For the other techniques it showed an average reduction of 11.9% in data transfer times. In general, the techniques that featured larger blocks had shorter data transfer times because fewer iterations were needed per frame to transfer data and run those algorithms.

Cat-tail DMA performs better than the baseline because it provides a low-overhead software mechanism to manage data transfers. It employs circular buffering to maximize the block sizes stored in SPU local storage while accommodating concurrent transfer and processing on the cores. Also, by performing the data transfers in much larger block sizes, the communication overhead between SPUs is minimized. This also provides longer processing periods during which the shared bus is available for other cores to schedule and perform data exchanges.

Staggering the execution cores results in a fixed one-time latency applied to the execution time of the program. This charge is a small fraction of a single block transfer and is insignificant in the context of the several transfers that are required to process a single frame. Also, typical programs process several frames during execution. Also, the reduction in overall execution times due to cat-tail buffering more than compensates for this charge. Figure 26 shows the performance of cat-tail buffering for multimodal mean for an increasing number of frames. The results show a steady reduction in total execution times as the number of frames is increased.

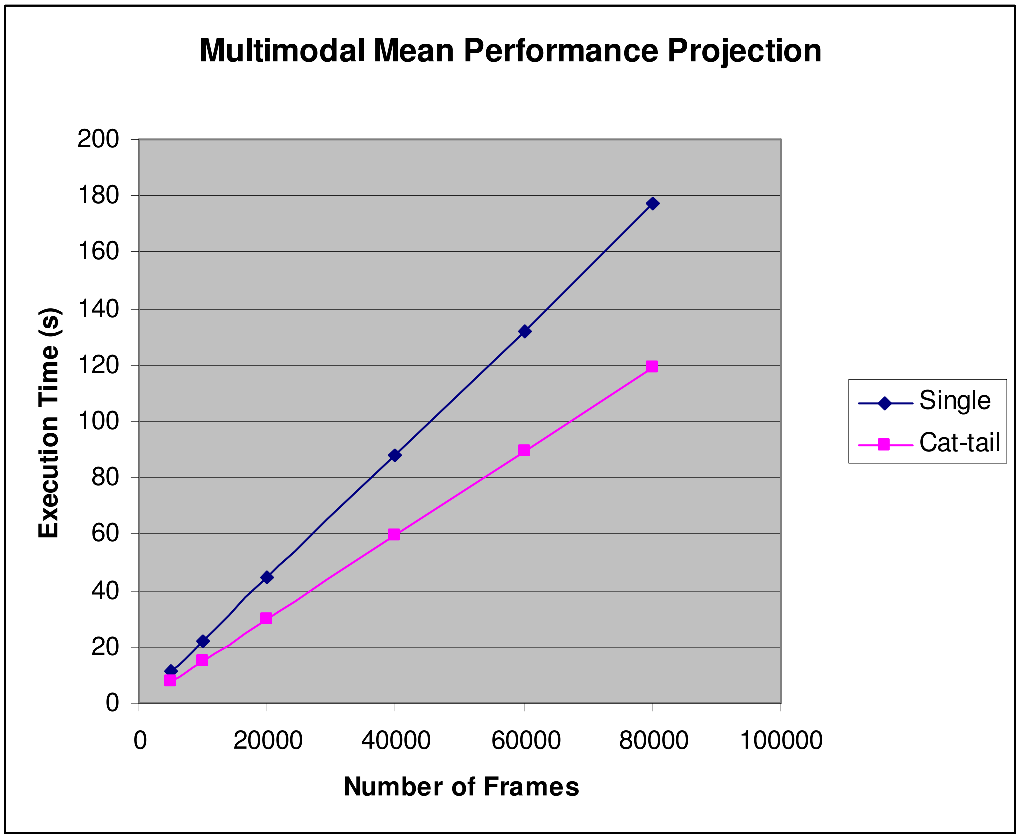


Figure 26 Performance improvements over time

This chapter introduced *cat-tail DMA* as a technique for efficient data transport for computer vision applications on multicore systems. Through experiments on the Cell BE this technique significantly reduces data-transfer times and in general overall processing times.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This dissertation explored techniques to efficiently map high performance early vision algorithms onto multicore embedded systems. Using the design of a pedestrian-tracking system for real-world embedded environments as a case-study it presented techniques to ensure high-performance of such a system. The systems approach encapsulated several components ranging from application software to hardware platforms. The first contribution presented an object modeling and tracking framework that minimizes computational and storage requirements of pedestrian-tracking applications while maintaining overall program accuracy. This framework was supported by a new background modeling technique called Multimodal Mean that provided fast and accurate segmentation of background/foreground content in embedded vision applications. Background modeling is the most expensive component of surveillance applications [21] and is responsible for up to 95% of surveillance workloads. Speeding up this component through Multimodal Mean algorithm results in a significant speedup of the overall software application

The second contribution involved optimizing early vision workloads on multicore embedded systems which are identified as ideal hardware platforms for executing pedestrian-tracking applications. Specifically, it addressed the very important issue of data reuse which helps reduce potential memory bottlenecks for early vision applications on multicore systems. It provided a technique for minimizing the number of data exchanges between the computing cores and main memory by introducing a temporal buffering step at the beginning of the processing. This resulted in a significant increase in the processing framerates of the application.

The third contribution further addressed the data-transfer bottleneck problem by exploring the actual scheduling of the data transfers by the individual cores. It presented a technique to minimize memory latency of image transfers on multicore processors by performing transparent, global DMA scheduling with concurrent program execution. With this technique, the data transfer times were significantly reduced as well as the wait times between processing of consecutive blocks of the image on each computing core.

All contributions were evaluated using experiments that were designed to be close to real-world scenarios using commercial-off-the-shelf components. Test sequences were carefully selected to feature real-world conditions and were collected with inexpensive devices such as webcams. In addition, the evaluations were performed on actual platforms rather than using simulators.

6.1 Summary of Results

The results from the contributions of this dissertation are summarized below:

6.1.1 Pedestrian-tracking application software

- Presented a fast, accurate object modeling framework [16], [18], and [19] that combines accurate background modeling with efficient object tracking.
    - Pedestrian-tracking application using this model ran at 0.78 fps when processing 640x480 pixel images on eBox 2300 Vesa PC.

  - Tracking accuracy of 92% for pedestrian-tracking application using this framework which is similar to other published results using other methods

6.1.2 Multimodal mean background modeling technique

- Developed *Multimodal Mean* [17],[30], a fast, accurate, background modeling technique targeted for embedded systems
  - MM method executes 6.2× faster than the MoG technique on the eBox 2300 Vesa PC.
  - MM method executes 4.23× faster than the MoG technique on the HP Slimline Pavilion platform
  - On both platforms MM requires 18% less storage per pixel and uses only integer operations.
  - MM provides comparable image quality and accuracy to MoG at the cost of other less accurate background modeling techniques

6.1.3 Fast, adaptive background modeling for multicore embedded Systems

- Presented a workload partitioning technique [52] to optimize the execution of background modeling algorithms on Multicore systems using temporal buffering
  - The technique results in a 25% increase in processing framerates for *Multimodal Mean*

  - The technique has very little overhead: a 0.023% increase in image decoding times and 125ms overall system delay for 320x240 images.

  - The technique reduces the number of image transfers by 50% for *Multimodal Mean*.

6.1.4 Fast, efficient image transport on multicore embedded systems

- Developed *cat-tail DMA*, a technique which provides globally ordered non-blocking DMA transfers on a multicore system.

  - This technique reduced data transfer times between main memory and processing cores by 32.8% for *Multimodal Mean* on the Cell B.E.

  - The technique increased utilization of core local storage by 60% over existing double-buffering techniques.

6.2 Future Work

In the future, contributions from this work will be extended to design a parallel, video surveillance driver for multicore systems. This driver will incorporate the ideas from the *Multimodal Mean* background modeling techniques into the segmentation component. Also, it will use the temporal buffering and cat-tail buffering techniques to improve processing framerates.

The current goal is to have an open-source driver for the commercially available Sony Paystation Eye webcam which will be extended for other devices at a later time. Individual SPU cores on the Cell B.E will be used to buffer and process different points of view which will be aggregated and displayed by the PPU. Currently, there are no known parallel vision platforms and this will represent a significant addition to the computer vision research community. With the emergence of multicore platforms such a tool will encourage development of parallel algorithms to leverage the computing resources available. Also, the driver can provide real-time benchmark suites for architects to fine-tune hardware designs.

The second area of future work is to continue with the development of real-time versions of other computer vision algorithms specifically targeted for embedded multicore systems. Particular areas of interests include 3D processing, video compression, and robotics/artificial intelligence.

REFERENCES

[1] Yilmaz, A., Javed, O., and Shah, M., “Object Tracking: A Survey,” *ACM Computing Surveys,* Vol. 38, No. 4, Article 13, pp. 1-45, December 2006.

[2] Aggarwal, J. K. and Cai, Q, “Human motion analysis: A review,” *Computer Vision and Image Understanding* Vol. 73, No. 3, pp. 428–440, March 1999.

[3] Gavrila, D. M., “The visual analysis of human movement: A survey,” *Computer Vision and Image Understanding* Vol. 73, No. 1, pp. 82–98, Jan. 1999.

[4] Moeslund, T. and Granum, E., “A survey of computer vision-based human motion capture,” *Computer Vision and Image Understanding*, Vol. 81, No. 3, pp. 231–268, March 2001.

[5] C.R. Wren, A. Azarbayejiani, T. Darrel, A.P. Pentland, “Pfinder: Real Time Tracking of the Human Body,” *IEEE Trans. PAMI*, vol.19, no.7, 1997.

[6] I. Haritaoglu, D. Harwood, and L. S. Davis. “W4: realtime surveillance of people and their activities,” *IEEE TPAMI*, 22(8):809–830, 2000.

[7] Ramanan, D., Forsyth, D. A., Zisserman A., "Strike a Pose: Tracking People by Finding Stylized Poses," *Proc. CVPR* 2005.

[8] L.M. Fuentes, S.A. Velastin, “Tracking-based event detection for CCVT systems,” *Pattern Analysis Application*, vol. 7, pp.356-363, 2005.

[9] O.T. Dapos, M. Leo, P. Spagnolo, P.L. Mazzeo, N. Mosca, M. Nitti, “A Visual Tracking Algorithm for Real Time People Detection,**”** *IEEE WIAMIS 2007.*

[10] T. Zhao, R. Nevatia, “Tracking Multiple Humans in Crowded Environment,” *Proc. of the 2004 IEEE Computer Society Conf. on Computer Vision and Pattern Recognition.*

[11] C. Sminchisescu, and B. Triggs, “Kinematic Jump Processes for Monocular 3D Human Tracking,” *Proc. CVPR*, 2003.

[12] C. Bregler and J. Malik. “Tracking people with twists and exponential maps,” *Proc CVPR*, pages 8–15, 1998.

[13] M. Isard and J. MacCormick. Bramble: “A bayesian multipleblob tracker,” *Proc ICCV*, pages 34–41, 2001.

[14] H. Sidenbladh, M. J. Black, and D. J. Fleet. “Stochastic tracking of 3d human figures using 2d image motion,” *Proc ECCV*, 2000.

[15] P. Viola, M. Jones, and D. Snow. “Detecting pedestrians using patterns of motion and appearance,” *Proc ICCV*, 2003.

[16] Apewokin, S., Valentine, Bales, R., B., Wills, L., Wills, S., “Tracking Multiple Pedestrians in Real-Time Using Kinematics,” IEEE *Embedded Computer Vision Workshop (ECVW08)* June 2008.

[17] Apewokin S., Valentine, B., Wills, S., Wills, L., and Gentile, A., “Multimodal Mean Adaptive Backgrounding for Embedded Real-Time Video Surveillance,”

Embedded Computer Vision Workshop (ECVW07), in Proceedings of CVPR 2007 June 2007.

[18] Valentine, B., Apewokin, S., Wills, S., Wills, L., and Gentile, A., "Midground Object Detection in Real World Video Scenes," *IEEE Conf. on Advanced Video and Signal Based Surveillance (AVSS07),* Sept. 2007.

[19] Valentine, B., Choi, J., Apewokin, S., Wills, S., Wills, L., "Bypassing BigBackground: An Efficient Background Model for Embedded Video Surveillance," *IEEE Int. Conf on Distributed Smart Cameras,* Sept. 2008.

[20] Silicon Integrated Systems Corp., "SiS55x Family Datasheet", Rev 0.9, 14 March 2002.

[21] T.P. Chen, H. Haussecker, A. Bovyrin, R. Belenov, K. Rodyushkin, A. Kuranov, V. Eruhimov, "Computer Vision Workload Analysis: Case Study of Video Surveillance Systems", *Intel Tecnology Journal* 2005.

[22] Cheung, S. and Kamath, C. "Robust techniques for background subtraction in urban traffic video," *Video Communications and Image Processing*, Volume 5308, pp. 881-892, SPIE Electronic Imaging, San Jose, January 2004.

[23] C. Stauffer and W.E.L Grimson, "Adaptive background mixture models for real-time tracking", *Computer Vision and Pattern Recognition*, pp 246-252, June 1999.

[24] Radke, R.J., Andra, S., Al-Kofahi, O., Roysam,B., "Image change detection algorithms: A systemic survey," *IEEE Trans. on Image Processing,* 14(3) pp. 294-307, March 2005.

[25] Piccardi, M., "Background subtraction techniques: a review," *IEEE International Conference on Systems, Man and Cybernetics*, Vol 4., pp. 3099-3104, October 2004.

[26] Toyama, K., Krumm, J., Brummitt, B., and 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.

[27] N. McFarlane and C.Schofield, "Segmentation and tracking of piglets in images," *Machine Vision and Applications* 8(3), pp. 187-193, 1995.

[28] Jabri, S., Duric, Z., Wechsler, H., and Rosenfeld, A., "Detection and location of people in video images using adaptive fusion of color and edge information," *IEEE International Conference on Pattern Recognition,* pp. 627-630, vol 4., September 2000.

[29] Appiah, K., Hunter, A., "A single-chip FPGA implementation of real-time adaptive background model," *IEEE International Conference on Field-Programmable Technology*, pp. 95-102, December 2005.

[30] Apewokin, S., Valentine, B., Forsthoefel, D., Wills, S., Wills, L., and Gentile, A., "Embedded Real-Time Surveillance Using Multimodal Mean Background Modeling," *Advances in Pattern Recognition, Embedded Computer Vision, Springer 2009.*

[31] DMP Electronics Inc., "VESA PC eBox-2300 Users Manual", September 2006.

[32] Silicon Integrated Systems Corp., "SiS55x Family Datasheet", Rev 0.9, 14 March 2002.

[33] M. Gschwind, “The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor”, *International Journal of Parallel Programming.*, vol. 35, no 3, pp. 233 – 262, June 2007.

[34] P. Kongetira, K. Aingaran, K. Olukotun, “Niagara: a 32-way multithreaded Sparc processor”, *IEEE Micro*., vol. 25, no. 2, pp. 21 – 29, March-April 2005.

[35] C. McNairy, R. Bhatia, “Montecito: a dual-core, dual-thread Itanium processor,” *IEEE Micro*., vol. 25, no. 2, pp. 10-20, March-April 2005.

[36] M. B. Taylor, et al., “The raw microprocessor: a computational fabric for software circuits and general purpose programs,” *IEEE Micro.*, vol. 22, no. 2, pp. 25-35, March -April 2002.

[37] K. Sankaralingam, 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.

[38] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, Scott Rixner, "Imagine: Media Processing with Streams," *IEEE Micro*, vol. 21, no. 2, pp. 35-46, Mar/Apr, 2001.

[39] T.G. Lane. Using the IJG JPEG Library. Independent JPEG Group, 6b edition, March 1998.

[40] Cell Broadband Engine Programming Tutorial http://www-01.ibm.com/chips/techlib/techlib.nsf, 2nd January, 2009.

[41] J. Fritts, “Multi-level memory prefetching for media and stream processors,” in *Proc. Int. Conf. Multimedia Expo (ICME)*, 2002, pp. 101–104.

[42] R. Cucchiara, A. Prati, and M. Piccardi, “Improving data prefetching efficacy in multimedia applications,” *Multimedia Tools Appl.*, vol. 20, no. 2, pp. 159–178, June 2003.

[43] C. Xia and J. Torrellas, “Improving the data cache performance of multiprocessor operating systems,” in *Proc. 2nd IEEE Symp. High-Performance Comput. Arch. (HPCA)*, 1996, pp. 85–94.

[44] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *Proc. Int. Symp. Comput. Arch.*, 1990, pp. 363–373.

[45] C. Zinner, W. Kubinger, “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 2006.

[46] Gene A. Frantz, Kun-Shan Lin, Jay B. Reimer, and Jon Bradley. "The Texas Instruments TMS320C25 Digital Signal Processor," *IEEE Micro*. Vol. 6, No. 6, pages 10-28, December, 1986

[47] D. Kim, R. Managuli, Y. Kim, “Data cache and direct memory access in programming mediaprocessors,” *IEEE Micro*, vol. 21, no. 4, pp. 33-42, July-Aug. 2001.

[48] M. Dasygenis, et al., “A combined DMA and application-specific prefetching approach for tackling the memory latency bottleneck,” *IEEE Transactions on Very Large Scale Integration Systems (VLSI),* vol. 14, no. 3, pp. 279 – 291, March 2006.

[49] T. Chen, D. Budnikov, C. Hughes, Y.-K. Chen, “Computer Vision Workloads on Multi-Core Processors: Articulated Body Tracking”, *ICME 2007*, Beijing, China, July 2007.

[50] M. Gschwind et al., "A Novel SIMD Architecture for the Cell Heterogeneous Chip Multiprocessor," *Hot Chips 17,* Aug. 2005.

[51] Yellow Dog Linux v 5.0. http://www.terrasoftsolutions.com/, 2nd January, 2009.

[52] Apewokin, S., Valentine, B., Choi, J., Wills, L., Wills, S., “Real-Time Adaptive Background Modeling for Multicore Embedded Systems”, *Journal of Signal Processing Systems, Springer 2008.*

[53] M. Kistler et al., "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro,* May/June 2006, pp. 10–23.

[54] L. M. Ni, K. Y. Wong, D. T. Lee, and R. K. Poon, "A microprocessor-based office image processing system," *IEEE Trans. Comput.*, vol. C-31, pp. 1017-22, Oct. 1982.

[55] C. Lee, T. Yang, Y-F. Wang, “Partitioning and scheduling for parallel image processing operations,” *Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, p.86, October 25-28, 1995.

[56] L.S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, “ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance**,”** *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing* 1996 Page(s):5 – 5.

[57] N. Zhang, C.-H. Wu, "Study on adaptive job assignment for multiprocessor implementation of MPEG2 video encoding," *Trans. on Industrial Electronics,* vol. 44, no. 5, pp. 726-734, Oct. 1997.

[58] CPU Road Map 2008: Maxing Out Moore's Law, *PC Magazine* http://www.pcmag.com/article2/0,2817,2222910,00.asp

[59] ARM limited. White Paper: Architecture and Implementation of the ARM Cortex-A8 Processor. *http://www.arm.com/products/CPUs/ARM Cortex-A8.html.*