Journal of Mobile Multimedia, Vol.6, No.2 (2010) 170-184

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

SENYO APEWOKIN, BRIAN VALENTINE, LINDA M. WILLS, SCOTT WILLS

Georgia Institute of Technology, U.S.A.
{senyo, bvalent, linda.wills, scott.wills}@ece.gatech.edu

Received August 12, 2009
Revised January 30, 2010

The emergence of multicore platforms has tremendous potential for achieving real-time performance of complex computer vision algorithms. However, these applications must run on embedded, mobile platforms with stringent size weight, power, and cost constraints. High utilization of local storage on execution cores and low-latency, high-bandwidth data transfers between this storage and main memory are critical for real-time mobile system performance. General purpose processors employ hardware techniques, such as high-speed bus architecture and efficient data arbitration schemes, to address the memory bandwidth gap. However, these techniques are insufficient for mobile systems requirements. Concurrent algorithmic and architectural optimizations are necessary. This paper uses concurrency to minimize data transfer latency when executing video surveillance algorithms on multicore embedded architectures. It introduces cat-tail DMA, a technique that provides low-overhead, globally-ordered, non-blocking DMA transfers. Using this technique, data transfer latencies are reduced by over 30% for background modeling applications, while the local core storage utilization is increased by 60% over existing techniques.

Key words: DMA, background modeling, embedded computer vision, mobile vision systems, multicore, parallel processing

1 Introduction

Demand for efficient image processing on non-traditional mobile computing platforms is being fueled by the proliferation of portable multimedia devices, such as cell phones, gaming systems, media players, and automotive imaging systems. The combination of low-cost imaging chips and high-performance, multicore, embedded processors heralds a new era in portable vision systems. Computer vision algorithms have the potential for highly data-parallel execution. However, in a mobile environment, an implementation must operate within the constraints of embedded systems, including low clock rate, low-power operation with limited memory.

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 overall program performance. On multi-core systems where several cores need to be furnished with data, the importance of efficient image transport is further magnified.

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 direct memory access (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 gains access to the bus during contention. In addition, the arbiter determines when source and destination transfer paths do not 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 multi-core architectures. Partitioning of the multimedia workload to most efficiently utilize all the cores is critical to overall system 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 must 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 multi-core embedded systems because of the limited amount of local storage available on each execution core. For mobile image and video processing applications which involve very large datasets, limited local storage means only 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 an entire frame. The collective concurrent execution on the individual cores can potentially 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 number of DMA collisions and can result in higher data transfer latency.

This paper evaluates single-buffered DMA and explores the potential to hide data-transfer latency on multi-core embedded systems through double-buffered DMA. It presents a new technique, called *cat-tail DMA,* which addresses the primary shortcomings of double-buffered DMA and provides low-overhead, globally-ordered, non-blocking DMA transfers on a multi-core system. With this method the wait times for data transfers are reduced by over 30% for background modeling applications while increasing the utilization of core local storage by 60% over existing techniques. The paper focuses on background modeling processes because background/foreground segmentation and background model maintenance typically dominate the execution and storage costs of video processing workloads [4].

This paper is organized as follows. Section 2 describes background work and highlights existing DMA optimizations. Section 3 compares single and double-buffered DMA and highlights the particular challenges with these techniques with respect to image processing on multicore embedded systems. Section 4 presents *cat-tail DMA* which addresses the challenges described and Section 5 evaluates it on a multicore platform. Section 6 concludes the paper.

2 Related Work

The hierarchy of computer vision applications is diverse. High-level algorithms tend to process relatively compact and persistent data sets of abstract objects. This processing closely resembles other applications that run on today's general purpose processors. In contrast, early vision tasks (typically at the pixel level), e.g., background modeling, edge detection, and image enhancement, exhibit high levels of data parallelism and feature little data reuse. Conventional general-purpose architectures have cached-based memory systems that are tuned for data reuse and hence are ill-suited for image processing [8]. In [19] an evaluation of a cache-based system and a direct DMA system is performed on the TI TMS320C6416 DSP [6] and the results show superior performance for the DMA configuration. A similar evaluation is performed with a MAP 1000 processor in [9] and the experiments yielded similar results.

Multicore processing is the primary performance driver for today's and tomorrow's high performance processors. Embedded processor development is following this approach with low cost, low power multicore platforms (e.g., ARM Cortex A9 MPCore [3], TI TMS320C6472 six core fixed-point digital signal processor [16]). These architectures are well-suited for computer vision algorithms because they provide hardware support for concurrent execution of these highly data-parallel workloads. Early vision algorithms often include pixel-level or local region-based computation that can be partitioned and mapped onto multiple cores. However, speculative execution, branch prediction, and other latency-reduction techniques that are commonly employed in general-purpose multicore processors are ill-suited for embedded mobile vision applications. Automatic data caching is especially troublesome, since data access patterns are rarely captured with traditional caches. To achieve efficient, real-time execution of data hungry, early vision algorithms, architectures must support explicit (program-controlled) data memory transfer and scheduling.

Many research efforts focus on optimizations and frameworks for efficient and programmer-friendly DMA transfer. In [12], a technique is presented that permits low overhead, user-level DMA without compromising protection and without requiring changes to the underlying operating system kernel. A fast Application Programming Interface (API) that integrates standard DMA tasks and manages a command list is proposed. This interface allows for the programming of embedded multiprocessors on FPGAs without complex DMA controllers. In [17], a series of DMA experiments is used as datapoints to derive a mathematical formula for achieving optimal performance of an algorithm on a multicore platform. Various other optimizations for DMA transfer are presented in [5], [11], [13], and [18]. These techniques typically involve streamlining and integrating DMA tasks and providing memory integrity for user-level DMAs. This simplifies the data transfer for application programmers and allows seamless interaction between main memory and execution cores. Also, the optimizations presented involve predicting the optimal block sizes for DMA transfers for various applications.

However, important challenges still exist for efficient DMA transfer on multicore embedded systems. The presence of several cores executing the same program concurrently means there is high probability that the cores will request data through DMA transfers simultaneously. A low-overhead scheduling scheme that minimizes collisions and provides contention-free scheduling periods on execution cores will help reduce the DMA transfer latency. Also, on multicore embedded systems where local storage is limited, several data exchanges between main memory and execution cores may

be necessary during program execution. By maximizing the amount of data processed during each exchange, the frequency of exchanges is reduced and subsequently the data latency due to collisions. As a result, ensuring high utilization of local memory is necessary while allowing for concurrent transfer and processing of data on execution cores.

This paper presents *cat-tail DMA*, a technique that provides low-overhead scheduling of DMA transfers on multicore systems while maintaining high utilization of core local storage.

3 Single vs Double-Buffered DMA

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 sometimes required in more extreme cases.

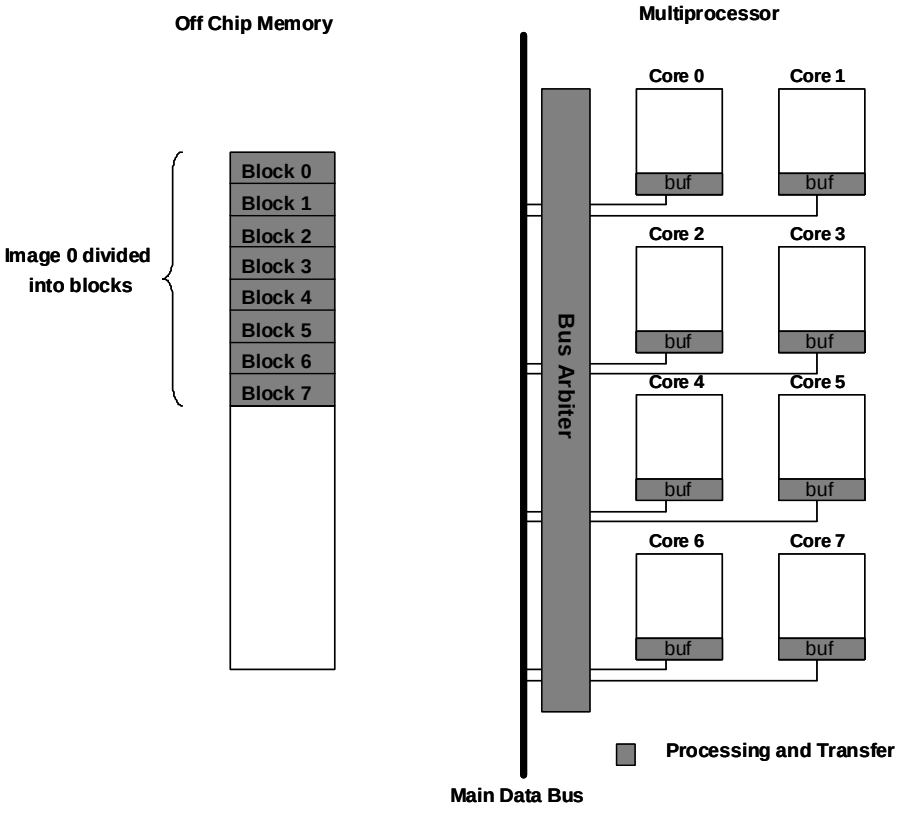

Figure 1 Single-Buffered DMA

Figure 1 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.

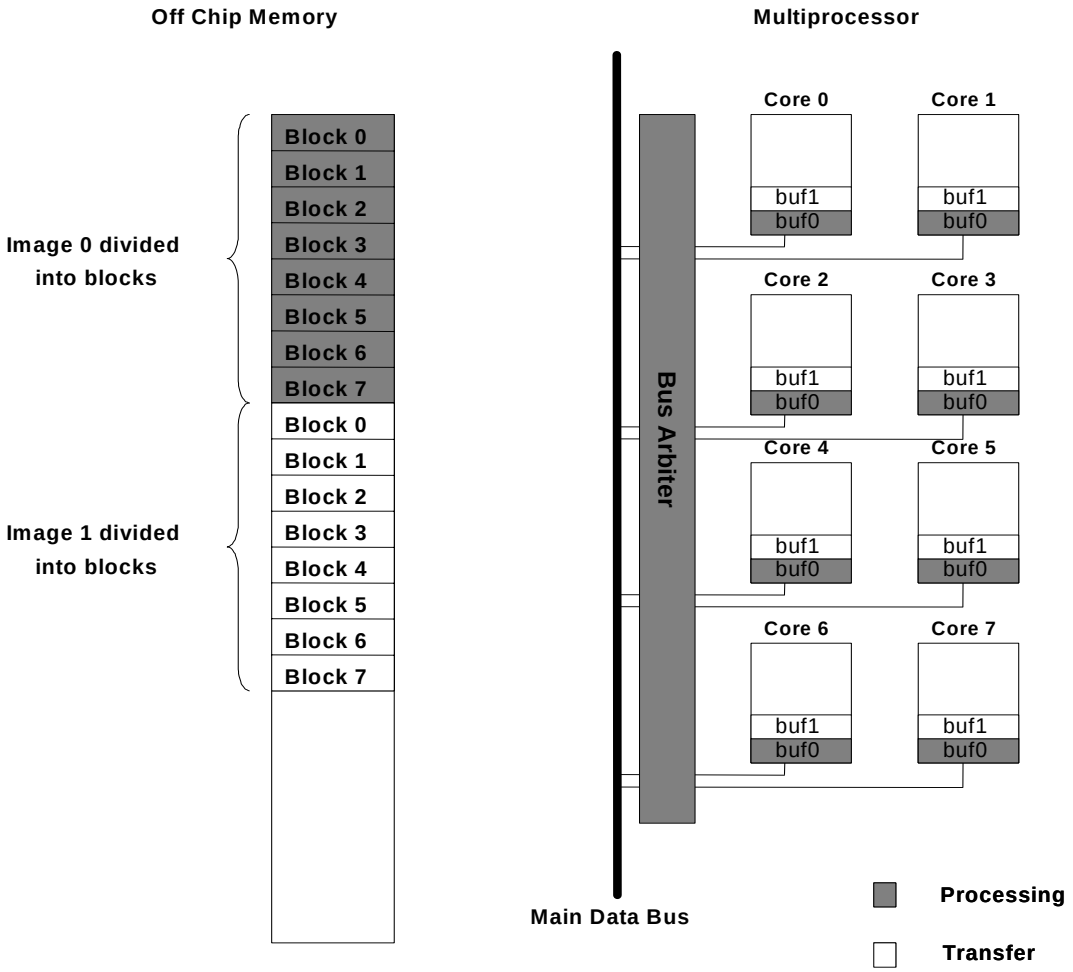

Figure 2 Double-Buffered DMA

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 2 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 read is hidden by overlapping the execution of a given block with the transfer of the next.

Two buffers, each of which can store a block of the image, are allocated in 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 on each core *c* are as follows.

To simplify the discussion, we will assume all cores working collectively can fully process one image in a single iteration. In other words, an image frame can be divided into equal-sized blocks among the cores and a single DMA transaction is required to transfer a block. Let *N* be the number of images to process.

```
Read image 0 block c into buffer 0
Write tag 0
For i in 1,2, … N:
    b = i&1 (i.e., b = 1 if i is odd, else b=0)
    Poll status of tag b
     Read image i block c into buffer b
    Write tag b
    Poll status of tag ~b
    Process buffer ~b
    Write image i-1 block c from buffer ~b
    Write tag ~b
Write image N block c from buffer 1
```

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 and so there is minimal waiting if any. Similarly there is some loop overhead between the write and the next read which again minimizes the wait time.

For computer vision applications, such as background modeling, double-buffered DMA presents challenges for multi-core embedded systems. In background modeling, a reference background model is computed that captures the features of the scene that are not moving or changing (e.g., a wall or sidewalk) or that have nonsalient movement (e.g., fluttering leaves). Typically, these applications compare the current image to the background model to subtract out the background and isolate the moving foreground objects. In this process, both the reference background model and the current image must be present on the cores for processing. As a result, these applications require the transfer of multiple sets of data for each processing iteration and require explicit management by the processor to ensure correctness.

To understand these issues, we experimented with background modeling on the Cell Broadband Engine [7]. This is a heterogeneous multicore chip which features one PowerPC (PPU) computing core and eight Synergistic Computing (SPU) cores on the same die. The PPU 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 SPU is a 128-bit RISC processor with 128 128-bit registers and 256 KB of local storage. The SPUs are designed for high-performance, data-streaming, and data-intensive computation. DMA is the primary method of communication between the SPUs 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 PPU handling program and data management and flow control, while the SPUs perform the pixel-level image operations. However, the limited on-chip SPU local storage presents challenges for efficient image data transport between main memory and SPUs [10].

Table 1 shows block configurations partitioned to store the image and background model for representative classes of well-known background modeling algorithms [2], [14], [15] on the Cell multicore embedded platform. Background modeling was chosen because it represents an important component of video surveillance algorithms and constitutes up to 95% of such workloads [4]. Also, it is memory intensive as well as computation intensive and serves as a good benchmark for image data transfer analysis. The configurations were chosen to maximize the utilization of processing core local storage on the Cell.

The Single Mode class in the table consists of algorithms that model pixels with a single background value (or "mode"), including frame differencing, approximated median, and weighted mean. The Sliding Window class includes algorithms that maintain a representation of the background based on the *w* images previous to the current image, where *w* is the sliding window size, (e.g., the background pixel value may be the median or mean value of that pixel location across the window). Mixture of Gaussians (MoG) [14] and Multimodal Mean [1], [2] are multimodal algorithms that maintain more than one background value for each pixel to model dynamic nonsalient motion in scenes (e.g., fluttering leaves may cause a pixel to vary between two or three different background values or modes). The multimodal algorithms were limited to 4 modes and the maximum-size of DMA transfers on the Cell B.E. is 16KB. Each SPU 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 1 SPU Maximum Block Transfer for Double-Buffered DMA

Algorithm	**Block Size (Pixels)**	**Image Size (Bytes)**	**Model Size (Bytes)**	**Number of Transfers**	
				Image	**Model**
Single Mode	19,200	57,600	57,600	4	4
Sliding Window	6,400	19,200	76,800	2	5
Mixture of Gaussians	800	2,400	80,000	1	5
Multimodal Mean	1,600	4,800	76,800	1	5

To process a given block for a technique such as MoG, a single DMA transaction is needed to transfer the image but five transactions are required to transfer the background model. Also, the entire background model must be available on the SPU to process the block. Similarly, the frame differencing technique requires four image transfer transactions and four background model transfer transactions.

The requirement for multiple data transfers of different datasets to complete one block of processing for computer vision applications differs markedly from traditional applications of double-buffered DMA where a single read/write transaction pair is overlapped with processes. This requirement 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 the potential collisions which can lead to high data transfer latency for entire block transfers. Furthermore, this problem can be exacerbated on multicore systems where several cores are executing the same program and making several simultaneous competing DMA requests.

Although this evaluation is performed on a single platform the results can be generalized across other multicore embedded platforms. These platforms feature limited on-chip local storage on execution cores and the same challenges apply.

4 Cat-Tail DMA

Cat-tail DMA addresses these issues 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 by:

1. Dividing the core processing of blocks into two phases: the *transfer and process* scheduling phase and the *process only* 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 key idea 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.

4.1 Core Processing

Two circular buffers (e.g., one for the image and the other for the background model) 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 3. The core processing is also divided into two phases in which unequal sized portions of the images are processed. In the first phase, *transfer and process,* a given block of the processing region is processed while the data in the transfer region is written out and new data read in. In the second phase, *process only,* the remainder of the processing region (which is composed of several contiguous blocks) 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.*

The example in Figure 3 shows the progression of both pointers along a circular buffer in each processing core. The process is the same for both circular buffers in each core. In circular buffer

implementations, a pointer is moved to the beginning of the buffer once it reaches the end as shown in the example.

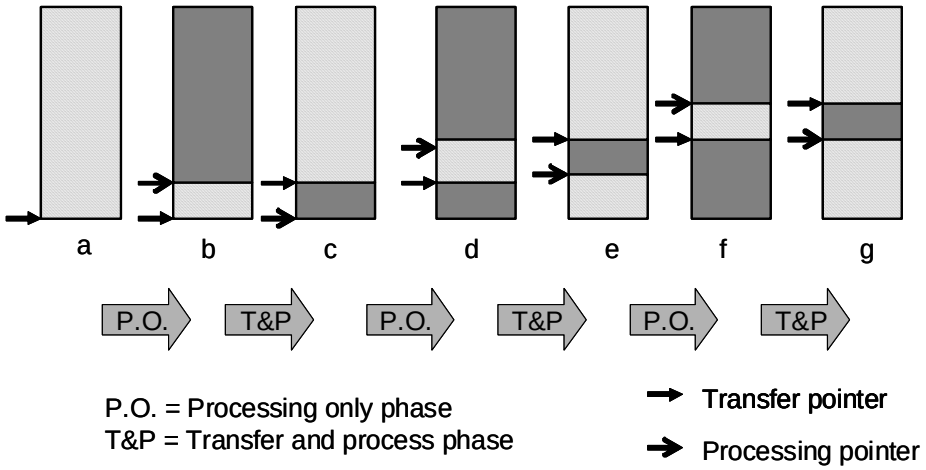

Figure 3 Circular Buffering

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 3a. The entire processing region is then processed in a *process only* phase (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 3b.

After initialization the two-phase processing occurs. In Figures 3c, 3e, and 3g the processing core is in *transfer and process* phase whereas it is in *process only* phase in Figures 3d and 3f. Figure 3c shows the first phase, *transfer and process*: a portion of the processing region is processed (dark area) concurrently with the exchange of data in the transfer region (light area). 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, *process only*, is shown in Figure 3d. The remainder of the processing region is processed (additional dark area) and the processing pointer is updated. Since there has been no transfer, the processed block from Figure 3c remains. Also, there is no data exchange during this phase and therefore the transfer pointer remains unchanged. In Figure 3e, the core reverts back to *transfer and process* phase, with the transfer and processing pointers pointing to updated regions. It proceeds to the *process only* phase in Figure 3f. The process is repeated in each core until the end of the program.

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. Figure 4 illustrates the staggered execution on the multicore system.

A token is passed around cores in a round robin fashion to signal which core is scheduled for data-transfer. During the first phase of processing, the processor on the core that possesses the token schedules all its data writes (e.g., image tile and background model) interleaved with the processing of the image pixels. The processor is not stalled to wait for completion of the transfer and the processing is continued. On the k^{th} iteration of processing in the phase, the processor checks for completion of the scheduled writes and proceeds to schedule reads. On the $2k^{th}$ iteration, the token is released to the next core and the process is repeated.

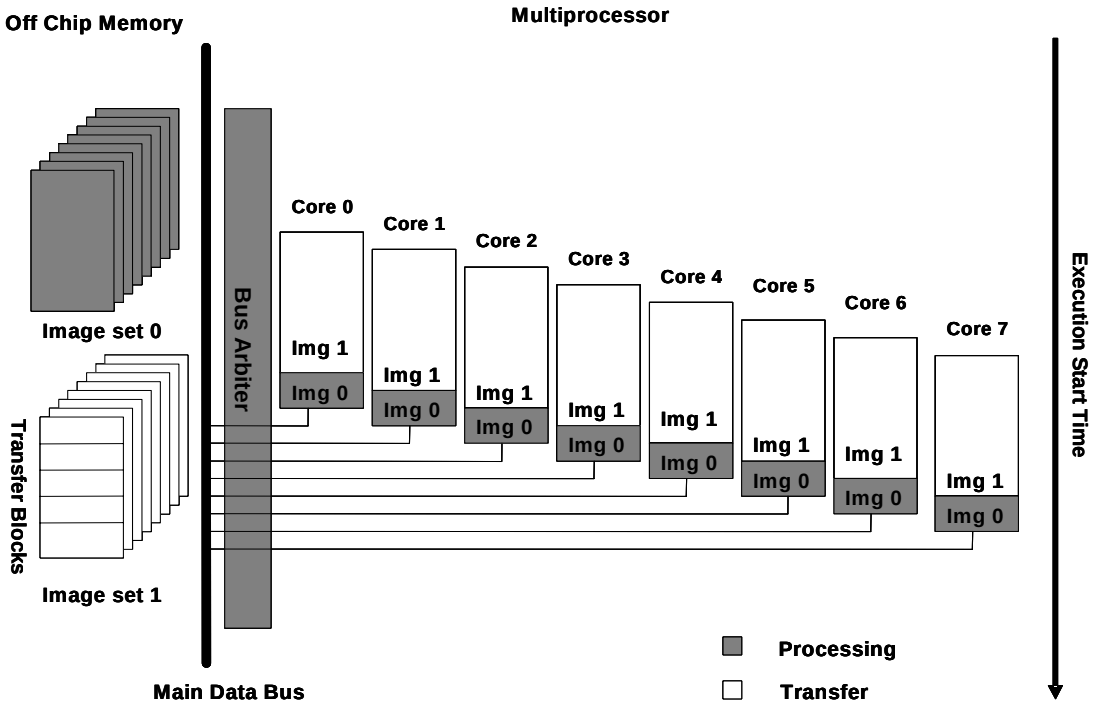

Figure 4 Staggered Execution

The parameter *k* is chosen such that all scheduled DMA transactions will be completed in the time taken to complete *k* pixel processing operations. On an embedded platform with limited memory, the number of DMA transfers required for complete utilization of local memory is several orders of magnitude less than the total number of pixels processed per block iteration. As a result, there is a wide range of possible values of *k*, to accommodate different image processing operations and execution platforms. However, for embedded platforms, DMA transfers should typically be completed during the processing of a given row of pixels, and for our experiments *k* was set to the image row size.

The execution is summed up as follows, where *CB1* and *CB2* refer to circular buffers 1 and 2 (e.g., the image tile buffer and the background model buffer in background modeling applications):

Phase 1 (Transfer and process):

Initialize process_counter
For each block A in CB1 transfer region:
Schedule write block A.
Write A's write_tag.
Process corresponding pixel location in CB1 and CB2's processing regions.
Increment process_counter.
For each block B in CB2 transfer region:
Schedule write block B.
Write B's write_tag.
Process corresponding pixel location in CB1 and CB2's processing regions.
Increment process_counter.
For i in process_counter … k:
Process (k-total_blocks_written) corresponding pixel locations in CB1 and CB2's processing regions.
Poll status of A and B's write_tags
For each block A in CB1 transfer region:
Schedule read block A.
Write A's read_tag.
Process corresponding pixel location in CB1 and CB2's processing regions.
For each block B in CB2 transfer region:
Schedule read block B.
Write B's read_tag.
Process corresponding pixel location in CB1 and CB2's processing regions.
*For i in k + process_counter … 2*k*
Process (k-total_blocks_read) corresponding pixel locations in CB1 and CB2's processing regions.
Release token.
*For i in 2*k … processing block:*
Process all remaining pixels first block of CB1 and CB2's processing block.

Phase 2 (Process only):

Process all remaining pixels in CB1 and CB2's processing regions.

Unique image read, background model read, image write, and background model 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 Evaluation and Results

An experiment was designed to evaluate the performance of cat-tail buffering on the Playstation 3 featuring the Cell B.E. For this experiment, the data structures for the background modeling algorithms were initialized by the PPU and held in main memory. The resolution of the images is 640 x 480 pixels and at startup the images were decoded by the PPU and held in a buffer in main memory. 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.

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

Mailboxes were used to communicate between SPUs and the PPU and signals were used for inter-SPU communication [10]. This allowed DMA transfers of data to be performed with minimum intrusion and very little communication overhead.

Circular cat-tail buffering combines the desirable features of the single-buffered and double-buffered techniques. It features the larger processing block sizes associated with the single-buffered DMA technique with the concurrent processing and data transfer operation associated with the double-buffered technique. Table 2 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 double buffering shown in Table 1. For example, when executing Multimodal Mean, cat-tail DMA processes 2560-pixel-sized sub-images per iteration as opposed to 1600-pixel-sized sub-images for double-buffered DMA, as shown in Table 1. This is possible because the separate buffers reserved to transfer and process images are not of equal size. Also shown in Table 2 is the maximum-sized block that can be held on each core (which is the configuration for single buffering). It is clear that the utilization of core storage during processing of cat-tail buffering is very close to the optimum case of having a single region with no concurrency.

Table 2 Utilization and Buffering Execution Times

Algorithm	**Blocks Held (Pixels)**	**Blocks Transferred (Pixels)**	**Buffering Execution Times (s)**	
			Single Buffering	**Cat-Tail Buffering**
Single Mode	38,400	30,720	28.11	25.00
Sliding Window	12,800	10,240	51.67	42.94
MoG	1,600	1,280	144.09	132.55
Multimodal Mean	3,200	2,560	88.19	59.53

Our performance results also show significant improvements in execution time for *cat-tail DMA* over *single-buffered DMA*. This is due to the concurrent processing and transfer of images as well as the low overhead management and scheduling of transfers on a multicore system. Figure 5 and Table 2 (two rightmost columns) show the performance of both the single and cat-tail buffering techniques.

The cat-tail buffering technique shows a 32.8% reduction in total processing time over the baseline single buffering for multimodal mean. For the other techniques it shows an average reduction of 11.9%

in processing times. In general, the techniques that feature larger blocks have shorter execution times because fewer iterations are needed per frame to transfer data and run those algorithms.

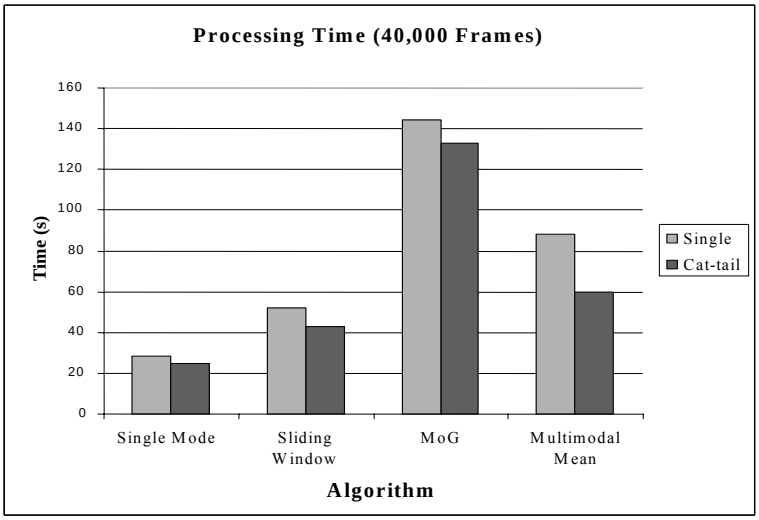

Figure 5 Performance of Buffering Techniques

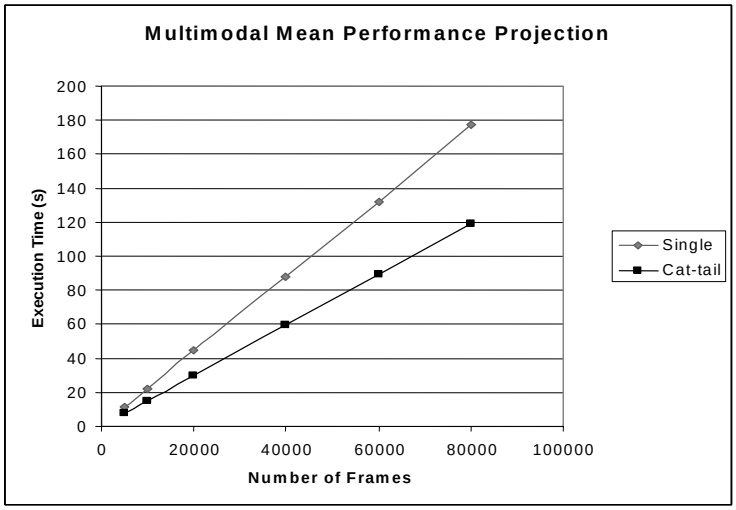

Figure 6 Performance Improvements Over Time

Cat-tail buffering 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 several transfers required to process a single frame and several frames during execution of the program. Also, the reduction in overall execution times due to cat-tail buffering more than compensates for this charge. Figure 6 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.

6 Conclusion

This paper introduces *cat-tail DMA* as a technique for efficient data transport for computer vision applications on multi-core systems. Experiments on the Cell BE show this technique significantly reduces wait times and overall processing times. Data transfer latencies are reduced by over 30% for background modeling applications, while the local core storage utilization is increased by 60% over existing techniques. High utilization of local storage on execution cores and low-latency, high-bandwidth data transfers between this storage and main memory are critical for real-time system performance on mobile platforms. This technique better supports early vision applications in embedded, low-power environments where slower clocks and voltage scaling normally reduce performance.

References

1. Apewokin, S., Valentine, B., Choi, J., Wills, L., and Wills, S., "Real-Time Adaptive Background Modeling for Multicore Embedded Systems," to appear in *Journal of Signal Processing Systems,* Springer, New York 2010.
2. Apewokin, S., Valentine, B., Forsthoefel, D., Wills, L., Wills, S., and Gentile, A., "Embedded Real-Time Surveillance Using Multimodal Mean Background Modeling," *Advances in Pattern Recognition, Embedded Computer Vision,* editors Kisačanin, B., Bhattacharyya, S., and Chai, S., Chapter 8, pages 163-175, Springer, London 2008.
3. ARM limited. White Paper: The ARM Cortex-A9 Processors. Available online (Aug. 31, 2009) at *http://www.arm.com/pdfs/ARMCortexA-9Processors.pdf*.
4. Chen, T.P., Haussecker, H., Bovyrin, A., Belenov, R., Rodyushkin, K., Kuranov, A., and Eruhimov, V., "Computer Vision Workload Analysis: Case Study of Video Surveillance Systems," *Intel Technology Journal*, Vol. 9, No. 2, (2005), 109-118.
5. Dou, Y., Deng, L., Xu, J., and Zheng, Y., "DMA Performance Analysis and Multi-core Memory Optimization for SWIM Benchmark on the Cell Processor," *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications, ISPA '08,* (2008), 170 – 179.
6. Frantz, G. A., Lin, K-S., Reimer, J.B., and Bradley, J., "The Texas Instruments TMS320C25 Digital Signal Processor," *IEEE Micro*. Vol. 6, No. 6, December (1986), 10-28.
7. Gschwind, M., et al., "A Novel SIMD Architecture for the Cell Heterogeneous Chip Multiprocessor," *Hot Chips 17,* Aug. 2005.

8. Khailany, B., Dally, W., Kapasi, U., Mattson, P., Namkoong, J., Owens, J., Towles, B., Chang, A., Rixner, S., "Imagine: Media Processing with Streams," *IEEE Micro*, vol. 21, no.2, Mar/Apr, (2001), 35-46.
9. Kim, D., Managuli, R., and Kim, Y., "Data cache and direct memory access in programming mediaprocessors," *IEEE Micro*, vol. 21, no. 4, July-Aug. (2001), 33-42.
10. Kistler, M., et al., "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro,* May/June (2006), 10–23.
11. Lin, K., Huang, C., and Lo, C., "Design and Implementation of a Schedulable DMAC on an AMBA-Based SOPC Platform," *IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2006,* December (2006), 279 – 282.
12. Markatos, E. and Katevenis, M., "User-level DMA without operating system kernel modification," *Third International Symposium on High-Performance Computer Architecture,* February (1997), 322 – 331.
13. Shida, S., Shibata, Y., Oguri, K., and Buell, D., "An optimization method of DMA transfer for a general purpose reconfigurable machine," *International Conference on Field Programmable Logic and Applications, FPL 2008,* September (2008), 647 – 650.
14. Stauffer, C. and Grimson, W. E. L., "Learning Patterns of Activity Using Real-Time Tracking," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. 22, No. 8, August (2000), 747-757.
15. Toyama, K., Krumm, J., Brummitt, B., and Meyers, B., "Wallflower: Principles and Practices of Background Maintenance," in *Proceedings of the International Conference on Computer Vision, ICCV 1999,* (1999), 255-261.
16. Loc Truong, "Low power consumption and a competitive price tag make the six-core TMS320C6472 ideal for high-performance applications," Texas Instruments, available online: focus.ti.com, October (2009), 1-7.
17. Tumeo, A., Monchiero, M., Palermo, G., Ferrandi, F., and Sciuto, D., "Lightweight DMA management mechanisms for multiprocessors on FPGA," *International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2008*, July (2008), 275 – 280.
18. Vivek, P., Jiang, W., Zhou, Y., and Bianchini, R., "DMA-aware memory energy management," *The Twelfth International Symposium on High-Performance Computer Architecture, HPCA 2006,* February (2006), 133 – 144.
19. Zinner, C., and Kubinger, W., "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)*, April (2006), 361-372.