In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, Dallas, Texas, March 2010. To appear.

BUFFER MANAGEMENT FOR MULTI-APPLICATION IMAGE PROCESSING ON MULTI-CORE PLATFORMS: ANALYSIS AND CASE STUDY

Dong-Ik Ko (d-ko@ti.com), Nara Won (won@ti.com),

DSPS Systems, ASP Division Texas Instruments, Dallas, TX 75243, USA

Shuvra S. Bhattacharyya (ssb@umd.edu)

Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, 20742, USA

ABSTRACT

Due to the limited amounts of on-chip memory, large volumes of data, and performance and power consumption overhead associated with interprocessor communication, efficient management of buffer memory is critical to multi-core image processing. To address this problem, this paper develops new modeling and analysis techniques based on dataflow representations, and demonstrates these techniques on a multi-core implementation case study involving multiple, concurrently-executing image processing applications. Our techniques are based on careful representation and exploitation of *frame*- or *block*-based operations, which involve repeated invocations of the same computations across regularly-arranged subsets of data. Using these new approaches to manage block-based image data, this paper demonstrates methods to analyze synchronization overhead and FIFO buffer sizes when mapping image processing applications onto heterogeneous, multi core architectures.

Index Terms- shared memory, multiprocessing, dataflow

1 RELATED WORK

Various previous efforts for reducing synchronization overhead in parallel processing environments have been reported. These techniques can be categorized into two groups — those based on compile-time scheduling techniques and those based on runtime scheduling. Runtime techniques may employ hardware acceleration logic to boost communication performance or specialized arbitration logic to handle dynamic changes in task priorities. These approaches provide reduced synchronization overhead, but increase power consumption, and also increase SOC design and validation costs due to requirements for specialized intellectual property (IP) blocks. On the other hand, compile-time techniques are more power- and cost-efficient, but require accurate estimation of execution times for computational and communication tasks.

Techniques in [1, 4, 8] exploit the structure of static schedules for iterative dataflow graphs, and reduce synchronization overhead by deriving minimal sets of synchronization operations that preserve the sequencing constraints imposed by the original dataflow graph and the given schedule. However, these techniques do not take into account parameterization of communication operations in terms of block size. In this paper, we integrate block size parameterization into communication analysis to provide a more general approach for synchronization optimization.

As a runtime technique, Monchiero et al. propose a hardware-assisted modeling of spin lock polling to reduce synchronization overhead [5]. This work carefully analyzes the effect of a hardware spin lock mechanisms on synchronization-induced contention for communication resources. However, this approach is modeled based on an assumption of unpredictable operational patterns among computational threads, and focuses more on general network-on-chip processing applications. In contrast, in this paper, we focus on the image processing domain, and develop compile-time techniques that exploit predictable behavior that is exposed by formal dataflow representations of the input applications.

For signal processing applications, the highest level data frames of blocks can often be divided naturally into lower-level blocks, which are processed through repeated sequences of block-based operations. Such multi-level block-structured data organization is particularly common in image and video processing applications. Ko and Bhattacharyya have developed techniques for formal modeling and quasi-static scheduling of such multi-level block processing applications by building on the framework of parameterized dataflow graphs [3].

In multi-level block processing scenarios, the lower level block sizes are significant factors in determining FIFO buffer sizes, and block processing throughput improvements due to vectorized implementation[6]. In this paper, we carefully integrate block processing with synchronization cost, and demonstrate the relevance of such integration for multi-core image processing systems. Our proposed approach can be applied as a post-analysis approach in conjunction with existing dataflow scheduling and resource allocation techniques, such as those developed in [2, 8]. Our approach can contribute also to early-stage design estimation of trade-offs between performance and buffer memory utilization.

2 SHARED BUFFER IMPLEMENTATION

The processing units can employ different memory management policies that are tailored towards the specific characteristics and contexts of the processing units. In our targeted platform (ARM+DSP), the ARM core provides an MMU (Memory Management Unit), and associated support for virtual memory.

In contrast, the DSP core provides direct access to physical memory. This allows for fast data token transfer within DSP memory space, but has limitations in terms of buffer memory protection and fragmentation. Shared memory regions must be handled with special care due to their synchronization requirements and larger access times. Determination of shared region buffer sizes is a critical factor influencing the efficiency of inter-core data token delivery. This paper addresses trade-offs between shared buffer configurations and inter-core communication performance.

We employ a circular buffering policy to map dataflow buffers onto shared memory regions. For inter-core communication,

synchronization functionality must be coordinated carefully with circular buffer management to provide correct, efficient memory transfer of data tokens between actors (functional node in a dataflow graph: similar to task/thread) that execute on different cores.

3 SCHEDULING FORMULATIONS FOR FRAME-BASED PROCESSING

Multi-media applications often process data streams in terms of frames of data that encapsulate contiguous sub-regions of the enclosing streams. The example in Figure 1 shows how the frame size influences the synchronization overhead for shared buffer regions. In this example, actor A is placed on the ARM core and actors B and C are placed on the DSP core. The communication channel between actors A and B is mapped onto a shared buffer region between the ARM and DSP cores. In contrast, the channel between actors B and C is placed in a non-shared buffer region associated with the DSP local memory space.

In Figure 1(b), the whole image is divided into four frames, which correspond to sub-images. Each iteration of the dataflow graph of Figure 1(a) processes a single sub-image, and therefore, four iterations of the graph are required to process a complete image. In the dataflow graph of Figure 1, annotations next to the actor ports show the numbers of tokens produced and consumed on each actor input and output port, respectively.

Given the frame-based image processing approach illustrated in Figure 1, the frame size (number of pixels in each sub-image), which we denote by N_{sub}, is given by the product $(W_{sub} \times H_{sub})$, where W_{sub} and H_{sub} represent the width and height of each sub-image, respectively. The image size (number of pixels in each complete image), which we denote by N_t, can be expressed similarly by the product $(\alpha_N \times N_{sub})$, where α_N represents the number of sub-images in each image.

The value α_N can be expressed as

$$\alpha_N = \frac{N_t}{N_{sub}}. \quad (1)$$

The application dataflow graph, which we denote by G_{sub}, processes N_{sub} pixels per iteration. .

In Figure 1(b) $\alpha_N = 4$. A valid schedule S_{sub} for processing a single frame can be expressed as

$$S_{sub} = (3(2A)B)(2C). \quad (2)$$

If S_{sub} is a valid schedule for processing a single frame, then a valid schedule for processing a complete image can be derived as the looped schedule

$$S_{N_t} = (\alpha_N S_{sub}). \quad (3)$$

Thus, for example, a valid schedule for the overall application represented by Figure 1(b) is given by

$$S_{N_t} = (4(3(2A)B)(2C)). \quad (4)$$

In Figure 1(c-d), the frame size N_{sub} is twice the value of N_{sub} for Figure 1(a-b), and thus, from (1), we have that $\alpha_N = 2$. Furthermore, since the actors A', B', and C' in Figure 1(c) are obtained by vectorizing actors A, B, and C, respectively in Figure 1(a), it can be verified that schedule of (2) is also a valid single-frame schedule for the overall application represented by Figure 3(d). Thus, from (4), we have that

$$S_{N_t} = (2(3(2A)B)(2C)) \quad (5)$$

is a valid schedule for Figure 1(d).

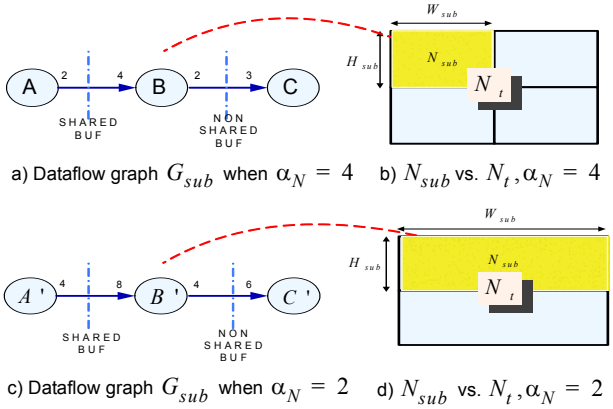

Figure 1. Impact of frame size on synchronization counts against shared buffer memory region.

4 MODELING SYNCHRONIZATION COST

Given an application dataflow graph, a multi-core target processor onto which the graph is to be mapped, and a shared buffer edge e in the graph, we refer to the **synchronization count** of e as the total number of synchronization operations that must be completed for reading and writing data on the edge when processing a complete iteration of the application graph. In our case study, an application graph iteration corresponds to the processing of an image frame (sub image).

We decompose the synchronization count metric into components $Synch_{write}(e)$ and $Synch_{read}(e)$, respectively, where the former(6) refers to the number of synchronization operations for writing data tokens into the associated shared buffer, and the latter(7) refers to the number of synchronization operations for reading.

$$Synch_{write}(e) = \text{LCM}(prod(e), cons(e))/prod(e), \quad (6)$$

and

$$Synch_{read}(e) = \text{LCM}(prod(e), cons(e))/cons(e), \quad (7)$$

where *LCM* denotes the *least common multiple* operator.

Here, for one synchronization write request associated with a shared memory edge e, $prod(e)$ tokens are written onto the corresponding shared memory buffer. This can be viewed as a vectorized writing of all of the output data for edge e that is associated with a single invocation of the source actor for e. Similarly, for one synchronization read request with a shared memory edge e, $cons(e)$ tokens are read from the corresponding shared memory buffer.

If E_{sh} represents the set of shared memory edges (i.e., the edges that are mapped to shared memory buffers), then the total number of synchronization operations required to process a dataflow graph iteration can be expressed as

$$Synch_{G_{sub}} = \sum_{e \in E_{sh}} Synch(e), \quad (8)$$

where

$$Synch(e) = Synch_{write}(e) + Synch_{read}(e). \quad (9)$$

The total number of synchronization operations required in the processing of a complete image can be expressed as the product of the number of image frames in a complete image and the synchronization count for a single frame:

$$Synch_i = \alpha_N \times Synch_{G_{sub}}. \quad (10)$$

For example, for Figure 1(a), the total number of synchronization operations per graph iteration can be derived from (6), (7), and (8) as

$$Synch_{G_{sub}} = \frac{\mathrm{LCM}(2,4)}{2} + \frac{\mathrm{LCM}(2,4)}{4} = 2 + 1 = 3, \quad (11)$$

and then the synchronization count $Synch_i$ for a complete image can derived from (10) as

$$Synch_i = 4 \times 3 = 12. \quad (12)$$

Similarly, for Figure 1(b), the total number of synchronization operations per graph iteration and complete image can be derived, respectively, as

$$Synch_{G_{sub}} = \frac{\mathrm{LCM}(4,8)}{4} + \frac{\mathrm{LCM}(4,8)}{8} = 2 + 1 = 3, \quad (13)$$

and

$$Synch_i = 2 \times 3 = 6. \quad (14)$$

5 ANALYZING SYNCHRONIZATION COST

The $T_{synch}(G_{sub})$ required for synchronization as a frame-level application graph (*frame processing graph*) G_{sub} processes an image frame (sub-image) is the total time taken for synchronization associated with processing an N_{sub}-pixel frame. This *synchronization time* can be estimated as

$$T_{synch}(G_{sub}) = (T_{stub}(G_{sub}) + T_{malloc}(G_{sub})). \quad (15)$$

$$T_{stub}(G_{sub}) = \delta_{G_{sub}} \cdot Synch_{G_{sub}}. \quad (16)$$

Here, $T_{stub}(G_{sub})$ represents the total synchronization set-up time (overhead due to common, synchronization "stub" code associated with inter-core communication) throughout execution of a single iteration (processing of a single frame) of G_{sub}. $T_{stub}(G_{sub})$ is independent of the frame size (N_{sub}), and depends on the total number of required synchronization operation count($Synch_{G_{sub}}$). $\delta_{G_{sub}}$ depends on bus architectures and synchronization methods. The term $T_{malloc}(G_{sub})$ represents the time taken to process a buffer allocation request from a shared buffer region. This term can be estimated as being directly proportional to N_{sub}. The value of $T_{malloc}(G_{sub})$ also depends on the profile of memory fragmentation in the shared buffer region at the time of the associated allocation request. This second factor — fragmentation-related overhead — is difficult to predict at compile time because multiple applications run simultaneously while influencing the run-time status of the shared buffer region.

During design space exploration, it can be useful to have N_{sub} bounded below by a minimum allowable frame size N_η, and to view N_{sub} as an integer multiple $n \cdot N_\eta$ of this minimum frame size. Here, if m represents the ratio N_t/N_η of the image size to the minimum frame size, then m and n satisfy

$$mod(m, n) = 0. \quad (17)$$

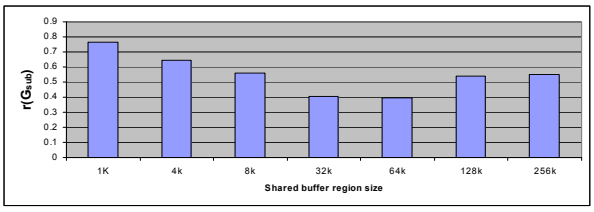

Figure 2. Synchronization overhead trends relating to data token transfer in shared buffer regions.

To analyze the synchronization performance of different frame processing configurations, it is useful to derive an estimate for the time T_η required to process a single iteration of G_{sub} if $N_{sub} = N_\eta$ — that is, if we use the minimum allowable frame size. Such an estimate can be derived as

$$T_\eta = T_o(G_\eta) + T_f(G_\eta) + T_{synch}(G_\eta), \quad (18)$$

where G_η represents the frame processing graph G_{sub} with $N_{sub} = N_\eta$; $T_o(G_\eta)$ represents the computational cost (required time) for processing of a single frame by G_η; and $T_f(G_\eta)$ represents the data token delivery time (excluding the time required for synchronization) for the FIFO buffers associated with the edges in G_η, and generally depends on the memory architectures employed in the target processor.

Given an arbitrary frame size (subject to (17)), the total time required to process the associated frame processing graph G_{sub} can be estimated as

$$T_{sub} = \frac{N_{sub}}{N_\eta} \times (T_o(G_\eta) + T_f(G_\eta)) + T_{synch}(G_{sub}), \quad (19)$$

and the total time to process a complete image using G_{sub} can be expressed as

$$\alpha_N \frac{N_{sub}}{N_\eta} \times (T_o(G_\eta) + T_f(G_\eta)) + T_{\mathrm{isynch}}(\alpha_N). \quad (20)$$

Here, $T_{\mathrm{isynch}}(\alpha_N)$ (the "complete-Image SYNCHronization time") represents the total synchronization time to process a complete image using α_N repeated iterations of G_{sub}. We model $T_{\mathrm{isynch}}(\alpha_N)$ by:

$$T_{\mathrm{isynch}}(\alpha_N) = \alpha_N \cdot T_{stub}(G_\eta) + \frac{T_{malloc}(G_i)}{\alpha_N}, \quad (21)$$

where G_i represents the frame processing graph that results from setting $\alpha_N = 1$ (i.e., setting the frame size to equal the image size).

$$T_{stub}(G_\eta) = \delta_{G_\eta} \cdot Synch_{G_\eta}. \quad (22)$$

$$Synch_i = \frac{N_t}{N_\eta} \times Synch_{G_\eta}. \quad (23)$$

(22) and (23) can be derived from (12), (14) and (16) in conjunction with G_η.

$Buf(G_{sub})$, which represents the total buffer size required to implement the frame processing graph G_{sub}, can be derived as

$$Buf(G_{sub}) = \sum_{e \in E_{nsh}} buf_{nsh}(e) + \sum_{e \in E_{sh}} buf_{sh}(e), \quad (24)$$

where E_{sh} and E_{nsh} represent the sets of dataflow graph edges that are mapped onto shared and non-shared buffer regions, respectively; $buf_{sh}(e)$ represents the buffer cost (memory requirement) of the individual shared buffer edge e; and $buf_{nsh}(e)$ represents the buffer cost of the non-shared buffer edge e. Because of the scaling of dataflow production and consumption rates as the frame size increases, we have for arbitrary G_{sub} that

$$B_\eta \le Buf(G_{sub}) \le B_i, \quad (25)$$

where B_η and B_i represent the total buffer size ($Buf(G_{sub})$) values, as given by (24), for the minimum and maximum frame sizes (η and N_t), respectively.

Building on the various evaluation metrics derived in this section, we can formulate the following ratio $r(G_{sub})$ as a figure of merit that characterizes the overhead of synchronization relative to the volume of data token transfer in a frame-based, multi-core image processing configuration:

$$r(G_{sub}) = \frac{T_{\text{isynch}}(\alpha_N)}{T_{\text{isynch}}(\alpha_N) + \frac{N_t}{N_\eta} \times (T_f(G_\eta))}. \quad (26)$$

The variation of $r(G_{sub})$ with candidate frame sizes and associated transformations of the frame processing graph is useful to take into consideration during design and implementation of a multicore image processing system.

6 EXPERIMENTAL RESULTS

This paper has developed methods for analyzing the impact of shared buffer regions on data transfer among different — homogeneous or heterogeneous — processing units in a multi-core platform. Our methods are based on analyzing the design space associated with alternative frame processing configurations, and include the derivation of a new figure of merit $r(G_{sub})$, which helps to characterize the synchronization performance associated with a given frame processing configuration.

Figure 2 shows the results of experiments that demonstrate our analysis — in particular the figure merit $r(G_{sub})$ — on the TI Davinci platform (TI DM6446). In these experiments, we applied the following set of three applications concurrently as part of our case study on multi-application, multi-core signal processing: an MPEG-4 decoder, an alpha blending application, and a JPEG decoder.

These results show that buffer synchronization overhead plays a significant role, especially for smaller buffer sizes. Figure 2 also shows that initially, the impact of buffer synchronization decreases as buffer size increases — this is because the data transfer time becomes increasingly significant compared to the time required for synchronization functions. However, beyond a certain level of buffer size, the impact of buffer synchronization starts increasing. The main reason for this increase comes from increased contributions associated with the T_{malloc} factor as the volume of buffer allocation requests over the shared buffer region increases.

Figure 3 shows how T_{malloc} varies in relation to buffer size. As shown in Figure 3, the AC which is the time to process buffer allocation requests, increases rapidly as buffer size increases under simultaneous operation of multiple applications. The RE represents the time to release allocated buffers.

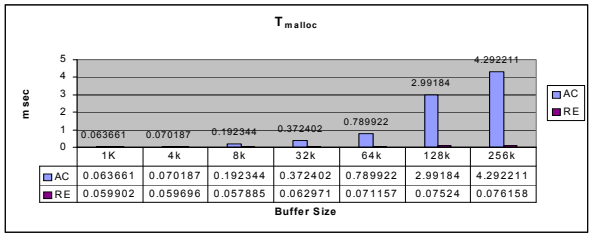

Figure 3. Buffer synchronization time(T_{malloc}) depending on shared buffer region size

Figure 4 shows how the data transfer rate (KBytes/sec) varies in relation to buffer size. A buffer size of 32KB provides the best transfer rate in our case study. This experiment quantifies how small buffer sizes cause high synchronization overhead because they effectively increase the frequency at which synchronization operations need to be carried out in conjunction with T_{malloc}.

7 CONCLUSION

As the complexity of multi-media embedded systems increases, heterogeneous multi-core platforms are increasingly attractive from an implementation perspective. This paper has analyzed the impact of buffer size, frame processing, and synchronization performance on overall system performance, and demonstrated this analysis with experiments that involved multiple, concurrent image processing applications DM6446.

Useful directions for further work include developing the combined optimization algorithm of (21), (24), and (26); developing tool support to help in automating the exploration approaches demonstrated in the paper.

REFERENCES

[1] S. S. Bhattacharyya and E. A. Lee. Looped schedules for dataflow descriptions of multirate signal processing algorithms. *Journal of Formal Methods in System Design*, pages 183-205, December 1994.

[2] P. Hoang and J. Rabaey, Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. In *IEEE Transactions on Signal Processing*, vol. 41, no.6, June 1993.

[3] D. Ko and S. S. Bhattacharyya. Modeling of block-based DSP systems. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 40(3):289-299, July 2005.

[4] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235-1245, September 1987.

[5] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Power/performance hardware optimization for synchronization intensive applications in MPSoCs. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2006.

[6] S. Ritz, M. Pankert, and H. Meyr. High level software synthesis for signal processing systems. In *Proceedings of the International Conference on Application Specific Array Processors*, August 1992.

[7] S. Ritz, M. Pankert, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application Specific Array Processors,* October 1993.

[8] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition, 2009.

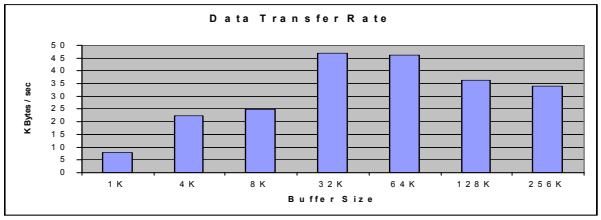

Figure 4. Data transfer rate as buffer size varies.