An Efficient Application Processor Architecture for Multi-Core Software Video Decoding

Chun-Jen Tsai, *Member, IEEE*, Yan-Ting Chen, and Chien-Chih Tseng

***Abstract*—In this paper, we propose a new multicore application processor architecture that facilitates the adoption of the fine-granularity software-pipeline parallelism without causing extra burden on the system bus. The proposed SoC architecture can simultaneously support the traditional symmetric multi-processor (SMP) multi-threading and the proposed software-pipeline applications efficiently. The programming model of the proposed architecture is compatible with the existing SMP operating systems. For the implementation of the pipeline-based parallelism, new programmer-friendly system calls are suggested to take advantage of the new software-pipeline datapath. The proposed architecture with four RISC cores is implemented on an FPGA development board for verification. An AVC/H.264 baseline profile video decoder that explores the pipeline parallelism with dynamic pipeline-stage partitioning is implemented on the target platform to justify the benefits of the proposed architecture. Experimental results show that the adoption of the proposed pipeline datapath architecture into existing application processors enables new potentials in exploring software parallelism.**

***Index Terms*—Multicore application processors, parallel video decoding, software pipeline architecture, wavefront video decoding.**

I. Introduction

THE SUPPORT for parallel decoding of a compressed video bitstream is an important application for multicore mobile application processors. As rich-multimedia applications keep pushing for new standards (e.g. HEVC [1]) and new capabilities (e.g., higher resolutions and frame rates), many early adoptions of the emerging applications have to rely on software-based decoder implementations. Although current state-of-the-art mobile application processors can handle the decoding task of a modern video content of 1080p at 30Hz on a single processor core [2], multicore parallel decoding would be inevitable as the resolution, bitrate, and frame rate become higher. Most video codecs have some design considerations for parallel encoding and decoding. For example, the slice structure is a well-known video codec tool for parallel encoding and decoding that has been supported virtually by all video coding standards since MPEG-1/2. Although slice-based parallel decoding can be supported efficiently for multicore application processors [3], there is no guarantee that such high-level structure exists for any particular video content.

A more general way to explore video decoding parallelism is to investigate the data dependency structure at the macroblock (MB) level or below. Before coding, each video frame is partitioned into video macroblocks. For most video codecs, an MB is composed of 16×16 pixels of the video data. It is the basic coding unit of most modern video codecs. The MB-level parallel decoding algorithm can be applied on existing mobile processors with the symmetric multi-processor (SMP) architecture [4][5][6]. However, it is essential that there is an efficient caching subsystem to handle massive parallel data accesses to the main memory. For parallel decoding below the MB level, the fine-granularity pipeline parallelism can be used [7][8]. The pipeline decoding architecture is usually adopted by the VLSI implementations of video codecs [7][9]. The concurrency behavior of a pipeline requires intensive communication between different pipeline stages. For ASIC's, this is usually not a problem since consecutive pipeline stages use dedicated high bandwidth interconnections, pipeline registers, or on-chip memory for communication. On the other hand, for software pipeline implementations on the mobile application processors today, the inter-stage traffics will put significant burden on the system bus because different cores can only share large amount of data through the main memory. Therefore, although the pipeline architecture is usually adopted for the VLSI design of video codecs, some researchers regard the pipeline model as impractical for software implementations when the granularity of a pipeline stage is small [6].

In this paper, we have proposed an innovative hardwired pipeline datapath for the SMP processors. The pipeline datapath is a special hardwired inter-processor communication (IPC) channel that allows each processor core to manipulate the data passing through it. The pipeline-like 'operate-and-transfer' of a data set across the IPC channel does not consume any system bus bandwidth. Furthermore, it is easy to design software to take advantage of the proposed pipeline datapath. With the proposed architecture, a parallel video decoder that explores the pipeline concurrency can be implemented efficiently. Furthermore, unlike the VLSI implementation of pipeline architecture, a software-based pipeline can easily adopt dynamic pipeline-stage partitioning to achieve better overall load balance.

The rest of the paper is organized as follows. In section II, we discuss the related work and present our contributions. The problem of parallel video decoding is formulated in section III. We also discuss the designs of the two parallel video decoders

This work was supported in part by the National Science Council, Taiwan, ROC under Grant NSC 100-2221-E-009-052-MY3.

The corresponding author Chun-Jen Tsai is with the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan (e-mail: cjtsai@cs.nctu.edu.tw).

that are used for experiments in this paper. In section IV, the architecture of the proposed pipeline datapath and the programming model is discussed. In section V, we conduct some investigations on the load-balancing issue of video decoding and verify how well the proposed architecture can deal with the variations of decoding complexity. In section VI, the performance of a dynamic pipeline-based parallel video decoder is compared against a wavefront-based video decoder on the proposed platform. Finally, in section VII, we present some discussions and future work.

II. Related Work and Contributions

A. Related Work

Software pipeline is a terminology widely used in different contexts in the field of computation architecture researches [10][11][12][13][14]. In this paper, we follow the software pipeline model defined in [12]. That is, a software pipeline is consisted of multiple ordered stages that process on a stream of input data. The output of one stage becomes the input to its direct successor. Each stage is an individual task consisting of a working set, program code, and task state.

Most of the studies on software pipeline try to find an efficient mapping of the pipeline behaviors to existing multiprocessor architecture [12][15][16]. For example, in [12], two runtime task-mapping schemes are proposed to create a dynamic software pipeline for a single-chip homogeneous computing cluster with proper network-on-chip (NoC) interconnects. The research in [16] points out a key issue of efficient software pipeline realization for variable-complexity data stream processing tasks on multicore processors. That is, the usage of scratchpad memory as the pipeline stage buffers. In this paper, we proposed a new hardwired logic that integrates the scratchpad memory into the IPC datapath. The architecture we propose has some similarity to the data-forwarding cache (DFC) in [17]. However, DFC is not designed for general-purpose software pipeline formation. Another interesting work that adopts software pipeline for parallel video decoding is presented in [8]. The study in [8] recognizes the overhead of inter-stage communication and synchronization when the task granularity of a stage is as small as a macroblock, thus, a frame-partition scheme is proposed to form a software pipeline with a larger task granularity for video decoding.

B. Contributions of this Work

The contributions of this work are summarized as follows:

1) We have proposed a new IPC datapath architecture for the existing SMP application processors such that a pipeline behavior can be executed efficiently without consuming extra system bus bandwidth for accessing pipeline stage buffers.
2) We have proposed a programming model for the new SoC architecture. The model is compatible with the POSIX multithread programming model [18] so that it is easy to adapt existing programs to take advantage of the new datapath.
3) We have implemented the complete hardware-software system on an FPGA to justify the benefits of the proposed architecture over existing solutions. The complete source code of the system, including the hardware implementation model of the SoC and the parallel decoder software, will be available in the public domain so that other researchers can duplicate and verify our results on an FPGA platform[1].

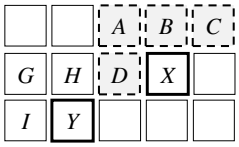

Fig. 1. The data dependency neighborhood of the MB decoding process. Decoding of the macroblock *X* requires the decoded data from the neighboring macroblocks *A*, *B*, *C*, and *D*. *X* and *Y* can be decoded in parallel.

III. The Parallel Video Decoding Problem

In this section, we discuss the problem of parallel AVC/H.264 video decoding for multicore processors. In general, a video codec will arrange the macroblocks of a video frame in the sequential scan-line order before coding. Each macroblock will be encoded as either an intra-predicted macroblock (i.e., an I-MB) or an inter-predicted macroblock. An inter-predicted macroblock can further be classified as either a single-hypothesis predicted macroblock (i.e., a P-MB) or a double-hypothesis bi-predicted macroblock (i.e., a B-MB). In this paper, we only study the parallel decoding behavior of the baseline profile of the AVC/H.264 standard [19]. Therefore, there are only I-MBs and P-MBs in the video bitstreams.

A. Parallel Video Decoding at the MB and Sub-MB levels

The MB neighborhood used to predict the content of the current MB is shown in Fig. 1. Assume that *X* is the current MB to be decoded, the decoded data of macroblocks *A*, *B*, *C*, and *D* will be used to predict the data of *X*. Thus, *X* cannot be decoded until the macroblocks *A*, *B*, *C*, and *D* are decoded. Similarly, decoding of *Y* depends on the results from the macroblocks *G*, *H*, *D*, and *I*. Nevertheless, there is neither direct nor indirect data dependency between the macroblocks *X* and *Y*, hence, they can be decoded in parallel. Although the MB-level data dependency pattern seems simple, it is not trivial to implement a scheduling policy that can achieve perfect load balance while minimizing the synchronization overhead [5].

One way to avoid the sophisticated MB-level scheduling problem is to explore the sub-MB level parallelism. The process of decoding a video MB is composed of several steps, as shown in Fig. 2 for the AVC/H.264 standard. A sequential stream of macroblocks must go through these steps one-by-one. If each step maps to a pipeline stage, at any given time instance after the initial delay, all stages will be concurrently processing different MB data. Although the pipeline architecture is usually adopted for ASIC's, it is often not used for software decoders. For example, in [4], it states that the main drawbacks of the pipeline parallelism for software implementations are the load balancing and scalability problems. The load-balancing

[1] The source code of the proposed four-core SoC and the video decoders will be available at http://www.cs.nctu.edu.tw/~cjtsai/research/pipeline_ipc/

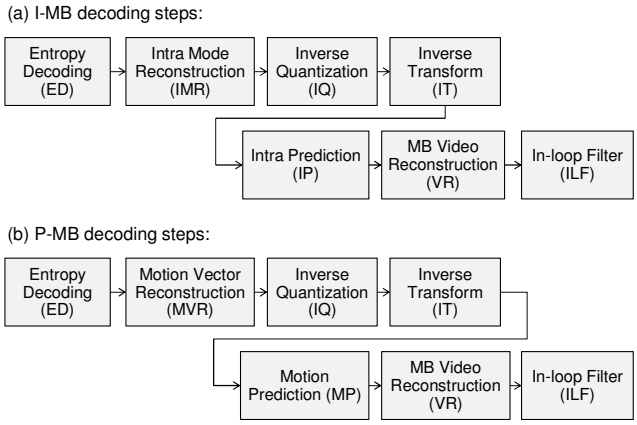

Fig. 2. The decoding steps of an I-MB and a P-MB of AVC/H.264. Note that the computations within some steps, such as the ILF, can be further divided into several finer steps to achieve better load balance.

problem is due to the fact that the decoding of a macroblock has variable complexity. If a software decoder tries to implement a decoding pipeline in a lock-step manner as in the VLSI implementation, a large portion of the decoding time will be wasted on synchronization. However, as we will show later, this issue can be solved by introducing a stage buffer between two pipeline stages to absorb the variation in complexity [16] and by applying dynamic pipeline partitioning.

The scalability of the pipeline parallelism is constrained by the number of the pipeline stages. As Fig. 2 shows, for an AVC/H.264 decoding pipeline, the decoding process of a macroblock has six or seven steps. Nevertheless, to reduce the inter-stage communication overhead and to achieve better load balance, most implementations will merge the decoding steps into three or four pipeline stages. On the other hand, if the inter-stage communication overhead is not a concern, it is possible to break down the seven (or six) steps into even more fine-granularity stages to achieve load balance. In fact, the data parallelism within a macroblock has less restriction than the data parallelism at the MB level. For example, the in-loop filter is one of the most computationally expensive steps of AVC/H.264. It takes sophisticated designs to perform parallel MB-level in-loop filtering [24][25]. However, within a single MB, the filtering operations of the 4×4 vertical edges (and, afterwards, the 4×4 horizontal edges) are parallel by design. Therefore, the ILF step in Fig. 2 can be further divided into several stages (two to four, for example) [25][26][27]. Since most mobile application processors will have no more than eight symmetric RISC cores, the scalability of the pipeline concurrency is not a major concern.

B. Software Pipeline-based Video Decoder

The software-pipeline video decoder we use in this paper divides the MB decoding process into three stages and let each processor core to execute one of the stages. The exact tasks that will be executed in the first two stages change according to the coding mode of the macroblock. The decoding pipeline for a non-skip mode P-MB is shown in Fig. 3. The pipeline for the I-MB is similar except that the first processor core executes ED + IMR and the second processor core executes IQ + IT + IP +

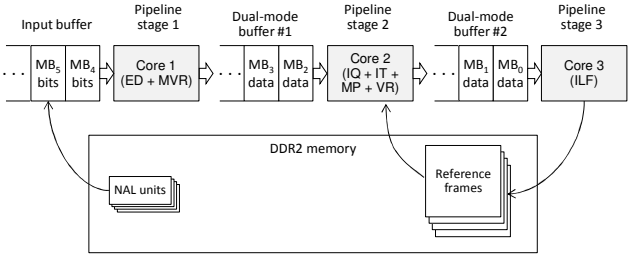

Fig. 3. A decoding pipeline for the P-MBs of AVC/H.264. The input buffer contains the compressed bitstream data (MB_i bits) of the video sequence. The stage buffer #1 and #2 contain the decompressed data structure (MB_i data) of each macroblock. Each node in the stage buffer #1 and #2 are 2KB in our implementation, including the reconstructed MB raw data. Note that the stage buffers also allow random accesses. They are not just FIFO buffers.

VR. Note that the VR process of a P-MB is composed of two parts, say, VR_1 and VR_2. The first part, VR_1 is composed of adding residual image to the motion predictor. The second part, VR_2, is composed of storing some pixel values in the local scratchpad memory for potential intra-prediction in the future. For a skip-mode P-MB, the first stage of the proposed decoding pipeline is composed of ED + MVR + IQ + IT + MP + VR_1 while the second stage is composed of VR_2 alone.

Inter-stage communication is the main problem of a software implementation of the decoding pipeline in Fig. 3. Unlike a hardwired pipeline that can keep most of its data accesses within the on-chip pipeline registers, current application processors require saving the intermediate data in memory banks (either an on-chip SRAM or an off-chip DRAM) through the system bus. Thus, parallel operations of all the decoding stages will create serious contentions on the system bus. To make things worse, the system bus is also used by other concurrent tasks and device controllers. As a result, a software-based decoding pipeline will not work well unless the pipeline stage buffers can be accessed without using the system bus bandwidth.

The variation in decoding complexity is another challenge for the software pipeline implementation. Unlike a VLSI decoder where the computation logic of each stage can always be parallelized to meet the worst-case specification, the software pipeline implementation is constrained by the processor core capabilities. Therefore, the two stage buffers in Fig. 3 must be large enough to absorb the decoding time variations between two pipeline stages. If, at any given time, one of the buffers underflows or overflows due to a long duration of decoding complexity variations, extra processor cycles will be wasted for synchronization. In section IV, we will propose a new multicore application processor architecture that addresses these two problems for the implementation of efficient software pipelines. The key idea is to hardwire a dedicated operate-and-transfer datapath with integrated buffers across all processor cores such that most pipeline operations along this datapath will not cause any traffic to the system bus.

C. Wavefront-based Video Decoder

Since existing application processors do not allow efficient implementation of the pipeline-based video decoder, an alternative is to adopt the wavefront parallel video decoding

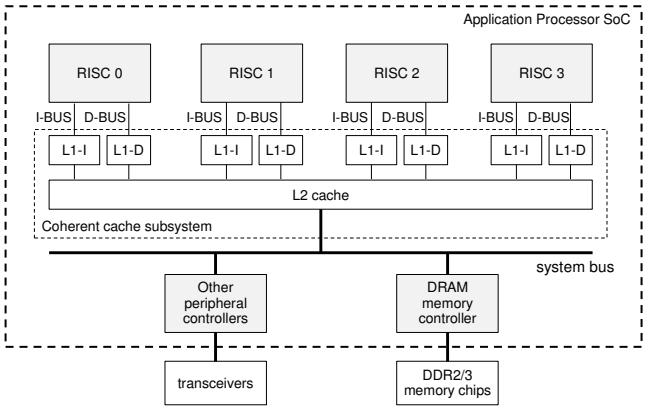

Fig. 4. The typical architecture of a multicore application processor. In this paper, we assume that the processor cores adopt the Harvard architecture and have separate instruction bus (I-BUS) and data bus (D-BUS). L1 and L2 are the level-1 and level-2 caches, respectively.

algorithm [4][5][6][28]. In this subsection, we present the key design parameters of a wavefront-based video decoder that will be used for comparison against the pipeline-based decoder on the proposed architecture. First, for entropy decoding, we adopt two different schemes. The first scheme is to let one of the cores to handle entropy decoding of the entire frame first, and then evenly distribute all the MB decoding tasks concurrently among all cores. This scheme is referred to as the non-interleaving wavefront decoder in Section VI. The second scheme is to let one of the cores to handle the entropy decoding task row-by-row. As soon as one row is entropy decoded, the MB decoding task will be dispatched to other cores. This scheme is called the interleaving wavefront decoder in this paper. For either type of the entropy decoding schemes, each MB decoding task is composed of the decoding of an entire row of MBs. Althoguh our scheduling policy seems quite simple, its actual performance is comparable to the more sophisticated scheduling policies published in previous work [5].

IV. The Proposed Application Processor Architecture

Multicore application processor has become one of the most crucial components of a mobile device such as the smartphones. The architecture of a typical four-core application processor is shown in Fig. 4. A multicore processor must integrate a cache subsystem to support thread-level parallelism effectively within a common address space [23]. The cache subsystem should provide independent instruction/data ports for all the processor cores so that the system bus will not be overwhelmed by the simultaneous instruction and data requests from all the cores. Furthermore, for programmer-friendly thread-level parallelism, the data cache should support coherent operations across the processor cores. If the cache controller does not enforce coherent cache operations in hardware, the programmer would have to synchronize the shared data manually across different threads, which is quite tedious. Even with a sophisticated coherent cache subsystem, the performance of a multi-thread application often may not scale up proportionally to the number of cores.

One way to relieve the burden of shared memory accesses of data intensive multi-thread applications from the system bus is

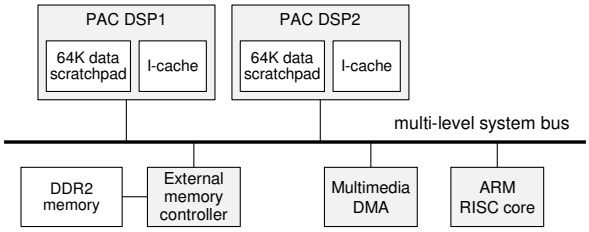

Fig. 5. The architecture of a PAC-Duo processor. Each DSP contains a scratchpad memory that is mapped to part of the global address space. Local accesses to the scratchpad do not consume the system bus bandwidth; but the external accesses to the scratchpad will go through the system bus.

to introduce local, scratchpad memory banks to each core [3][21][22]. For example, in the architecture (Fig. 5) of the PAC application processor [21], each DSP core has a local on-chip scratchpad memory of 64KB. To achieve the best performance for multi-thread applications, the thread running on each DSP core should put their working set of data in the scratchpad memory. Such DSP platforms are not designed for general-purpose multi-thread applications. The design is targeted for data-stream processing applications. If necessary, a data-stream processing task can be divided into two stages. Each data set must go through the two processing stages where each DSP core handles one processing stage. If DSP1 finishes the stage 1 subtask for a data set, the data set will be send to the scratchpad of DSP2's for stage 2 processing. On the PAC platform, the transfer of one data set from the DSP1 scratchpad to the DSP2 scratchpad will be performed by the DMA through the system bus. As a result, the operations may still consume a decent amount of system bandwidth.

In general, for smartphone devices, it would be better if a data-intensive application (e.g., a video decoder) uses as little system bus bandwidth as possible. Since a mobile device must handle sporadic man-machine interface (MMI) events with minimal delay, if a data-intensive multimedia application occupies the system bus most of the time, the MMI events may not be handled in time for smooth user experiences.

A. The Proposed Multicore Processor Architecture

Fig. 6 shows the proposed application processor architecture. The architecture is based on the traditional SMP architecture with an additional inter-processor communication channel. To support the proposed architecture, each processor core requires two independent set of instruction/data ports. The first set of instruction/data ports connect to the cache subsystem of the application processor and the second set of instruction/data ports connect to the local scratchpad memory through the IPC controller. The reason we connect the instruction bus to the scratchpad memory is for the storage of the boot code. The load/store unit of the processor core will direct a data access request to the corresponding bus according to the address of the request. This way, a processor core can access the data stored in its local scratchpad memory in one cycle through the IPC controller module. The detailed design of the IPC controller will be presented in section IV.*D*. When a software pipeline is mapped to the proposed architecture, there is a constraint on the mapping that a higher number stage has to be mapped to a

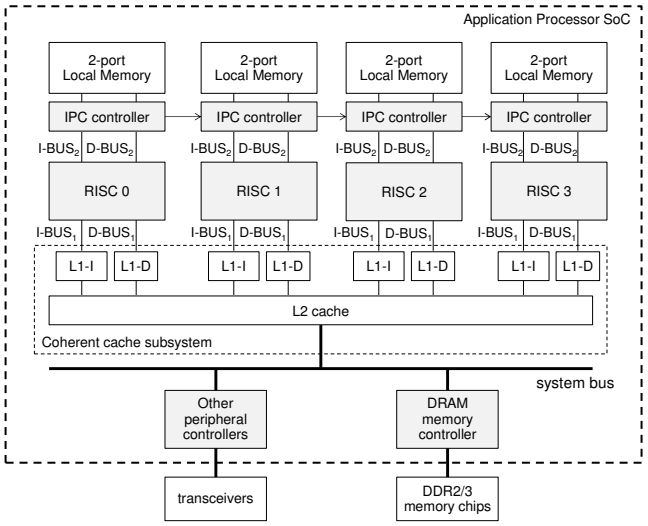

Fig. 6. The proposed multicore application processor architecture. The design facilitates heavy data-stream processing tasks across the IPC buses while leaving the system bus free for general-purpose tasks. More importantly, the design is programmer-friendly. Note that the links between the IPC controllers are unidirectional connections. This is a design choice that makes the IPC circuitry simpler but imposes some constraints on the assignment of the pipeline threads to the processor cores (see text for detail explanations).

higher number processor core because the data transfer direction of the IPC datapath is unidirectional. However, this constraint is merely a design choice to simplify the circuitry, not a theoretical limit.

B. The Programming Model of the Proposed Architecture

For a traditional SMP platform, a common address space is often shared by all threads. Such scheme simplifies the cache coherence and thread synchronization problems [21]. In addition, the POSIX thread programming model designed for such SMP architecture is well adopted by most operating systems (OS) and toolchains. Therefore, for the proposed new platform, if we can make its programming model as close to the POSIX model as possible, it would be much easier for the programmers to take advantage of the IPC datapath. In order to achieve this goal, instead of mapping the addresses of the four local memory banks to different areas in the common global address space, it would be better to map them to the same addresses.

For example, if the size of each local scratchpad is 64KB, then the address space from 0x0000 to 0xFFFF will always be mapped to the local memory regardless of which core it belongs to. In other words, when a processor core performs random accesses on an address between 0x0000 and 0xFFFF, it is accessing its own local scratchpad. Note that the resolution of any address in this range does not go through the global system bus decoder so there would be no interferences to the traffics on the system bus. With this address-mapping scheme, a "universal" API *lm_alloc(int size)* in the C programming language can be used by each processor core to allocate memory for data structures that are intended to be manipulated along the software-pipeline datapath. By "universal", we mean that a single copy of the thread-safe function codes (linked with the application program) is located in the DDR memory for all

```
void stage_k_thread(void *param)
{
    extern int *buf[];     // Shared stage buffer pointers.
    buf[k] = lm_alloc(64); // Each stage allocate its buffer
                           //   from its local memory.

    while (1) // never-ending pipeline operations
    {
        wait_for_new_data(buf[k]); // Wait for stage k-1's input.
        stage_k_operation(buf[k]); // Perform operations on buf[].
        wait_for_trans_ready(buf[k+1]); // Wait for stage k+1 ready.
        lm_transfer(buf[k+1], buf[k], 64); // Output to stage k+1.
    }
}
```

Fig. 7. The C pseudo code of a thread that performs stage *k* operations in the proposed pipeline datapath. The thread must be executed by a processor core in the correct position of the pipeline stages. The design here assumes a lock-step synchronized operation where each stage has a single-node buffer. For more efficient operations, each stage shall maintain a circular FIFO buffer. See text for detail explanations.

processor cores to fetch and execute, as in the case of the SMP architecture. The data structure allocated using *lm_alloc()* will reside in the corresponding processor's local scratchpad memory. Data processing operations performed on these data structures will not cause any traffic on the main system bus.

In order to pass a data set from one stage of the pipeline to the next, we suggest another API called *lm_transfer(void *dst_addr, void *src_addr, int len)*. This function can be used to transfer a block of data (with address *src_addr*) from the local scratchpad of processor *k* to the local scratchpad (at address *dst_addr*) of processor *k*+1. Again, the transfer of data using this function does not consume any data bandwidth of the main system bus. In addition, the physical data copy operations of this function is not performed by the processor core. Instead, it is performed by the IPC controller logic. The parameter *k* of the API is an implicit parameter. If the function is invoked by RISC 1 in Fig. 6, then *k* equals 1 and the transfer of data will be from the local memory of RISC #1 to the local memory of RISC #2. Note that the address range of the source data and the address range of the destination data are the same (both from 0x0000 to 0xFFFF), but they stand for different memory blocks. The implementation detail of the *lm_transfer()* function will be discussed in section IV.*D*.

The pseudo code of a thread that executes the pipeline stage *k* operations is shown in Fig. 7. Note that in this example, all pipeline stages are synchronized in a lock-step manner. Thus, each stage has only one single-node FIFO buffer (instead of a multi-node FIFO buffer as shown in Fig. 3). The array of buffer pointers **buf[]* is stored in the shared memory so that all cores have access to it. The function *wait_for_new_data(buf[k])* checks the stage buffer *buf[k]* and see if new data has arrived. The function *wait_for_trans_ready(buf[k+1])* awaits the ready flag (possibly a protected shared variable) from stage k+1 for data transfer to it. In short, each stage performs a two-step operation concurrently. In step one, the processor core performs random accesses to its local buffer *buf[]* for data processing. In step two, the IPC controller performs data transfer from its (manipulated) local buffer to the buffer of the next stage. It is important to point out that such lock-step operations do not handle variable complexity pipeline stages efficiently. For video decoding pipelines, each stage should maintain a circular FIFO buffer so that different pipeline stages

can operate asynchronously.

From the pseudo code in Fig. 7, one should notice a constraint with the proposed architecture. That is, the order of the pipeline stages of an application program must be preserved when the pipeline threads are assigned to the processor cores. For example, if a data-stream processing application performs three stages of operations on a sequence of data sets, we may create three threads to execute the three stages of operations, one for each stage. Since the IPC data path is unidirectional, therefore, the first thread that executes pipeline stage one can only be allocated to either RISC #0 or RISC #1. If stage one is allocated to RISC #1, then the threads for the pipeline stages two and three can only be allocated to RISC #2 and RISC #3, respectively. This constraint requires the application program to set the CPU affinity of each pipeline stage thread to some processor core. Although setting the CPU affinity of a thread is not a good idea in general, it is supported by the POSIX thread model. To enable flexible formation of the software-pipeline, the architecture requires a more sophisticated IPC controller. We will leave this investigation for future work.

C. *Realization of the Proposed Architecture on a Xilinx FPGA*

In this section, we present our implementation of the proposed architecture on a Xilinx FPGA platform. The Xilinx Microblaze processor IP is used for the RISC cores in the proposed architecture. Due to the limitation of the Microblaze IP, our physical implementation is slightly different from the architecture shown in Fig. 6. The key difference is due to the fact that the Microblaze processors do not support coherent data cache across multiple cores in an SMP organization. Therefore, the application program must explicitly call cache validation functions to ensure proper accesses of shared data. In general, manual validation of the data cache of a processor core will degrade its performance. However, we have designed additional hardware mechanisms so that synchronizations among the software threads do not rely on caching of the shared variables. According to our experiences, for the parallel video decoder, manual enforcement of the coherence of the data cache on the target platform causes little performance degradations. Fig. 8 shows the system architecture of our implementation of the SMP platform with the dedicated software-pipeline datapath.

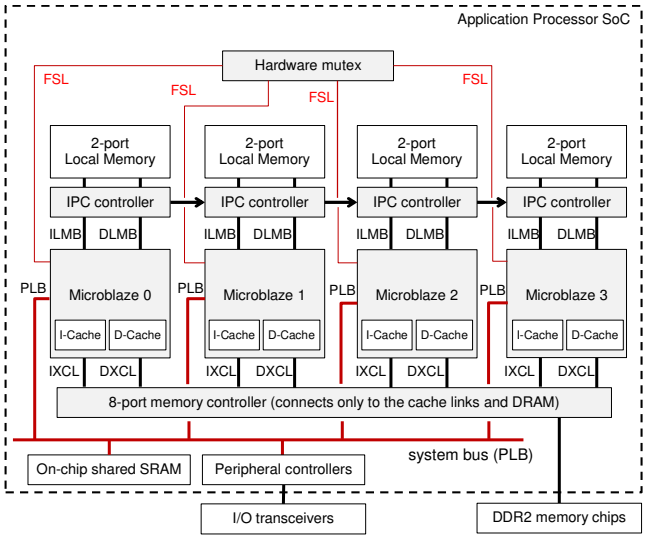

Fig. 8. The implementation of the proposed architecture on a Xilinx Virtex-5 FPGA. The two extra components, the hardware mutex and the on-chip shared SRAM are merely used for efficient synchronization among software threads. They are not essential parts of the proposed IPC datapath.

The Microblaze processor is a RISC core with the Harvard architecture. Each core has three sets of instruction/data ports. The first set connects to an on-chip memory bank via Local Memory Bus (LMB). The second set connects to the system bus, the IBM CoreConnect Processor Local Bus (PLB). The third set connects to the DRAM controller via the Xilinx Cache Link (XCL). In Fig. 8, the eight-port memory controller receives all the memory requests for the DDR2 memory chips directly from the XCL's, without consuming bandwidth of the system bus. The on-chip shared SRAM is exclusively used to store un-cached shared variables for thread synchronization.

In addition to the on-chip shared SRAM, we have designed a hardware mutex device for thread synchronization. The mutex device contains 64 32-bit special-purpose registers; each can be used as a mutex variable visible to all threads in an application program. Theoretically, one can implement a software-based mutex using the Peterson's algorithm [31][32] and the on-chip shared SRAM. However, the algorithm consumes system bus bandwidth and has higher latency than a hardware mutex that connects to all processor cores via dedicated buses, such as the Fast Simplex Links (FSL's). When a processor requests a mutex lock/unlock using the FSL bus, it only takes several clock cycles for the lock/unlock to take effect. Furthermore, when a processor core is blocked by a mutex lock request, the spin-lock loop of the request will only consume its exclusive FSL bandwidth. The design of the on-chip SRAM and the hardware mutex can significantly reduce the impact of the non-coherent data caches of the Microblaze cores for thread-synchronization. However, these IP's are not essential parts of the proposed pipeline datapath. On a system with a coherent data cache, they can be removed to simplify the SoC architecture.

D. *The IPC controller*

The key hardware component we propose in this paper is the pipeline-based IPC datapath. As shown in Fig. 8, the IPC data path mimics an operate-and-transfer pipeline. Each stage in the pipeline contains a processor core, an IPC controller, and a local scratchpad memory. The architecture of the IPC controller is shown in Fig. 9. The IPC controller handles the random memory accesses between the processor core and the scratchpad memory, and the burst data transfer between two consecutive pipeline stages. Ideally, for maximal performance, a four-port memory should be used for the scratchpad memory since there are four different memory requests that can happen simultaneously. The four types of requests include the write requests from the input IPC bus (IPCB_i) from the preceding stage, the read requests for data transfer through the output IPC bus (IPC_o) to the succeeding stage, the instruction fetch requests from the ILMB, and the random data read/write requests from the DLMB. However, a four-port memory is

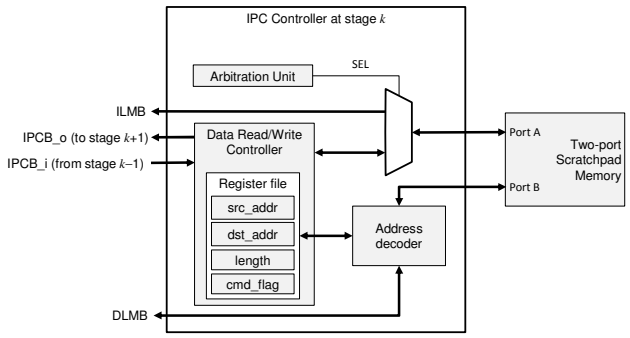

Fig. 9. The architecture of the IPC controller for the pipeline stage *k*. IPCB_o is the output IPC bus while IPC_i is the input IPC bus.

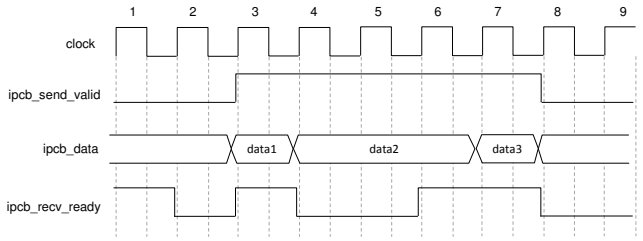

Fig. 10. The timing diagram of a transaction on the IPC bus. Each transaction transfers a burst of data words from the stage *k* scratchpad memory to the stage *k*+1 scratchpad memory. The data bus is 32-bit wide. By default, the sender will try to send a new word and the receiver will try to fetch a new word per clock cycle, unless one of them is has no access to its memory port.

expensive and may only increase the performance slightly.

In our implementation, we choose to use a two-port memory for the local scratchpad and let the ILMB and the two IPC buses share one local memory port. The DLMB gets exclusive usage of the other port since, at each pipeline stage, the processor core often performs intensive operations on the local data stored in the scratchpad memory.

Note that we do not store any application programs in the scratchpad memory. The scratchpad is only used to store the boot code and interrupt service routines (ISR's) for each core. Thus, the arbitration unit in Fig. 9 uses a simple yet effective arbitration policy to share a single scratchpad memory port among the ILMB and the two IPCB's. The ILMB has the highest priority of using the memory port because normally, at runtime, ILMB will only be used to execute ISRs that should be run as fast as possible. If the ILMB is not using the memory port, the two IPCB's will take turns sharing the memory port when contentions happen. It is important to point out that the transfer of data between pipeline stage buffers does not require processor's attention. The processor can perform data computations through the other memory port when the data transfer is in progress.

There are four 32-bit control registers in the IPC controllers that are used to trigger and specify the parameters of a burst of data transfer from the current stage scratchpad to the scratchpad of the next stage. These control registers are mapped to the addresses 0xFFF0 through 0xFFFC of the local memory address space. Any memory requests issued from the DLMB will be intercepted by the address decoder inside the IPC controller as shown in Fig. 9. If the address of the request matches one of the control register address, the memory operation will be directed to the IPC controller register file. Otherwise, it will be directed to the scratchpad memory. With this memory-mapping mechanism, the system function *lm_transfer(void *dst_addr, void *src_addr, int len)* can be implemented as follows. Upon the invocation of this function, the parameters *dst_addr*, *src_addr*, and *len* that specify the destination address, source address, and the length of the data transfer will be stored in the corresponding control register. Then, the *lm_transfer()* function will write a nonzero value into the *cmd_flag* register to physically trigger the burst data transfer. The IPC controller will transfer the two 32-bit registers, *dest_addr* and *length*, to the next stage IPC controller before transferring the actual data.

The timing diagram of the traffics over the IPCB for one transaction is shown in Fig. 10. The IPC controller will normally try to transfer one 32-bit word of data per clock cycle. However, since the IPCB_o and IPCB_i busses share the same memory port with the ILMB bus, the sender or the receiver may need to hold the transfer for a couple of cycles. Thus, the receiver drives an *ipcb_recv_ready* signal to tell the sender whether the current data will be fetched or not while the sender drives an *ipcb_send_valid* signal to tell the receiver whether there is a valid data on the bus in the current cycle or not.

E. The Software Kernel and Synchronization Schemes

In order to conduct multi-thread video decoding experiments on the proposed application processor, we have designed a thin operating system kernel that allows the four cores to cooperate in the SMP manner. The system kernel does not support preemptive multi-threading on a single core. Thus, each processor core can only execute one thread in the application program. Both the program code and the heap space are located in the DDR2 memory and shared symmetrically by all the processor cores. The stacks are private to each core and are located in its local scratchpad memory. The system memory map is shown in Fig. 11. When the system boots up, each processor core will execute its own boot code. The boot code of the first processor (Microblaze 0) jumps directly to the application program. The boot codes of the other processors (Microblaze 1, 2, and 3) sit in idle loops waiting for the application program to dispatch threads to them.

There are two ways to perform synchronization among threads (processors). First, there is a device in the system that contains 64 hardware mutexes. Each mutex can be locked or unlocked by a thread within a few clock cycles. If a thread tries to lock a mutex that has already been locked by another thread, it will enter a spin-lock loop that does not consume any system bus or data cache bandwidth until the mutex is unlocked by the owner thread. The second mechanism for synchronization is achieved through the on-chip shared memory in Fig. 8. Common flags and shared variables can be stored on the on-chip shared memory. The arbitration policy of the system bus is set to round robin such that no thread will be starving when accessing the shared data.

Although the write-through data cache of the Microblaze processor does not support coherent operations among multiple

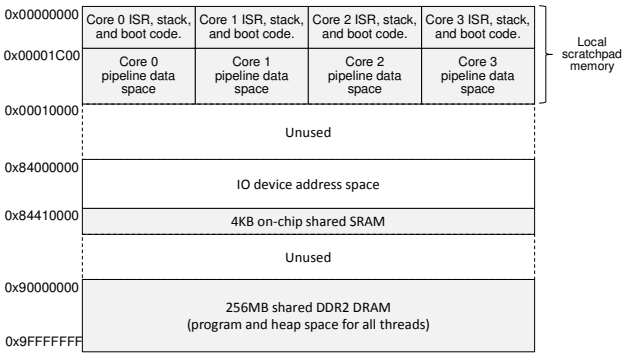

Fig. 11. The memory map of the system on an FPGA development board. The first 64 KB of the address space are not shared by all processor cores.

cores, the coherence of shared data can be achieved by data cache validation functions. For example, if a memory block in DRAM is modified by a processor, a different processor can invoke a function to force its data cache to re-cache the same memory block before it tries to access the data. Although such cache validation function can degrade the performance of general-purpose computing applications, it does not seem to cause obvious performance degradation for the parallel video decoding applications. In addition to the fact that our implementation does not rely on data cache for thread synchronization, another reason is that the data-sharing behavior of a parallel video decoder is unidirectional across processor cores.

V. Investigations on the Load Balancing Problem of the Pipeline-based Video Decoder

The performance of the pipeline-based video decoder is highly dependent on the balance of the computing load among pipeline stages. Since the decoding complexity of different video macroblocks varies significantly, it is not possible to use a lock-step pipeline with fixed stage partitions to achieve optimal load balance. That is why we propose to use dynamic pipeline partitioning plus a circular FIFO buffer between two pipeline stages to absorb the decoding complexity variations. The sizes of the pipeline stage buffers depend on the degree of variations of the MB decoding complexity. In this section, we conduct some experiments to investigate the relation between the macroblock modes, the pipeline buffer sizes, and the parallel decoding performance.

The measure we use for performance evaluation is the speedup ratio computed from dividing the decoding time of a three-stage (i.e., three-core) pipeline decoder by that of a single-core decoder. The speedup ratio is computed for every frame and used as an indication of the degree of parallelism of the proposed pipeline-based video decoder. Theoretically, the speedup ratio will always be less than three. However, in our implementation, the speedup ratio can be more than three mainly because the single-core decoder does not use the local scratchpad memory to store stacks (local variables) while the pipeline and the wavefront decoders do.

TABLE I
Macroblock Mode Distributions and Performance

Sequences	Bitrate	Intra MB %	Inter MB %	Skip MB %	speedup ratio
Crew	512 k	14.4	62.8	22.8	2.54
	1.5 m	15.4	76.5	8.1	2.63
Foreman	512 k	4.1	66.2	29.7	2.56
	1.5 m	4.2	80.4	15.4	2.74
Mobile	512 k	0.5	73.1	26.4	2.47
	1.5 m	0.5	86.4	13.1	2.72
News	512 k	1.4	29.2	69.4	2.31
	1.5 m	1.8	46.4	51.8	2.74
Stefan	512 k	3.0	59.4	37.6	2.29
	1.5 m	3.8	72.9	23.3	2.54

The percentages of the macroblock modes are computed for the whole sequence of 300 frames. The speedup ratio is obtained using a video decoder with the fixed pipeline stage partition shown in Fig. 3.

A. *Impact of the Macroblock Coding Modes on Performance*

The time it takes to decode a video macroblock depends on the coding modes. For example, it takes very little time to perform entropy decoding for a skipped macroblock. On the other hand, entropy decoding for an intra macroblock requires much more effort because intra macroblocks usually have more residual bits. TABLE I illustrates the distribution of the macroblock types for five MPEG test sequences. It is easy to notice that when the percentage of the skip MB becomes high, the speedup ratio drops. If the percentage of the inter MB becomes large, the speedup ratio becomes better.

The reason why the percentage of skip MBs affects the performance is quite simple. A sequence of consecutive skip MBs will spend very little time in the first entropy decoding stage. If the circular FIFO buffer depth between the first and the second stages is not large enough, shortly, the first stage decoder will be blocked and sit idly due to buffer full. Ideally, the circular FIFO buffer should be as large as possible, but in practice, its size is always limited by the size of the local scratchpad.

To get a better understanding of the impact of MB types on the parallel decoding performance, the speedup ratios of two

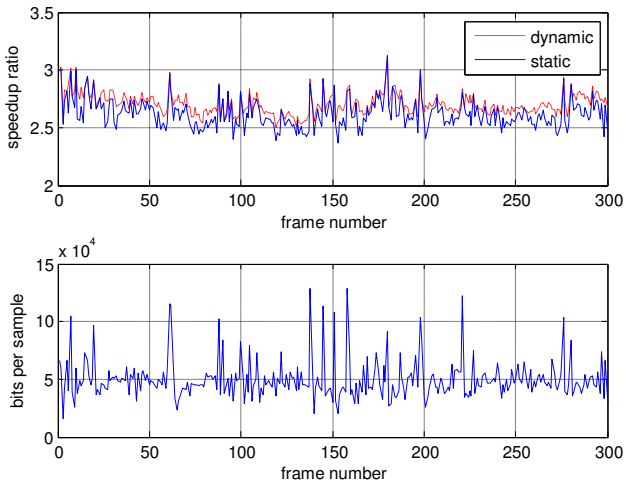

Fig. 12. Top: the speedup ratios of the dynamic and static pipeline partition decoders for Crew@1.5mbps. Bottom: bits per frame of the video sequence.

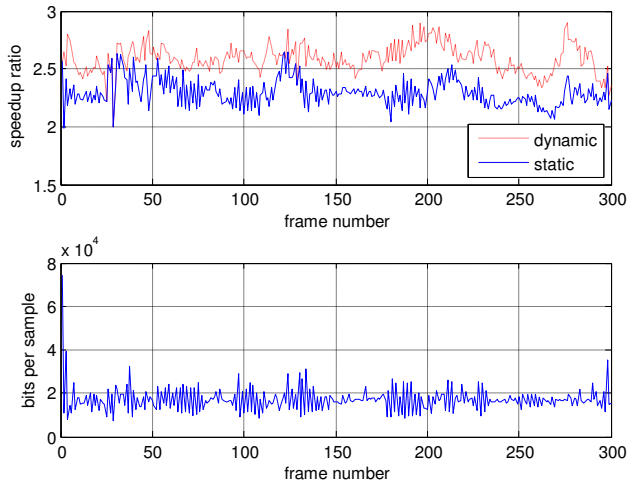

Fig. 13. Top: the speedup ratios of the dynamic and static pipeline partition decoders for Stefan@512kbps. Bottom: bits per frame of the video sequence.

software pipeline decoders, with static and dynamic pipeline partitioning schemes, respectively, for the 1.5 mbps Crew and 512 kbps Stefan sequences are shown in Fig. 12 and Fig. 13. In addition, the number of Intra 4×4 MBs of the Crew sequence, and the number of Skip MBs of the Stefan sequence are shown in Fig. 14. From Fig. 12, one can see that although the speedup ratios of the Crew@1.5 mbps sequence are above 2.5 for most of the frames, there are about a dozen frames (out of 300 frames) that have significant drop in decoding performance. Most of these drops correspond to sudden spikes in the numbers of bits per frame. The spikes in bits per frame also correspond to the peaks of the Intra 4×4 MB numbers shown in Fig. 14. On the other hand, for the Stefan@512 kbps sequence, the drops in the speedup ratios correspond to the spikes in the Skip MB numbers in Fig. 14.

B. Analysis on Circular Buffer Depth

In our implementation of the static pipeline partition decoder, there are three stages for the decoding of a macroblock. The first stage handles the entropy decoding and the Intra mode or motion vector predictions. The second stage handles the IQ, IT,

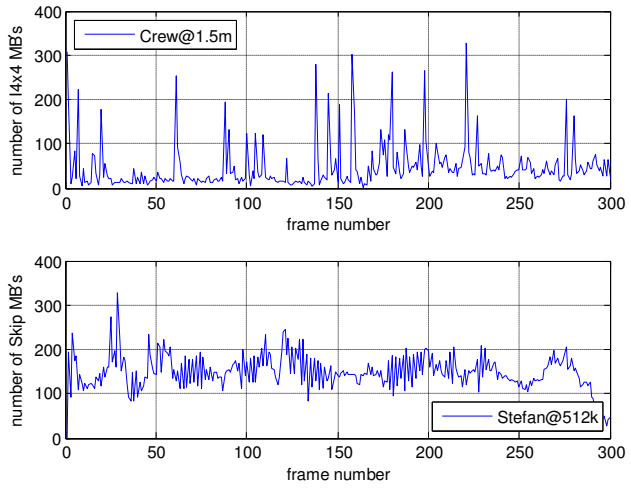

Fig. 14. The numbers of the I4x4 MBs of the Crew@1.5 mbps sequence and the numbers of the Skip MBs of the Stefan@512 kbps sequence.

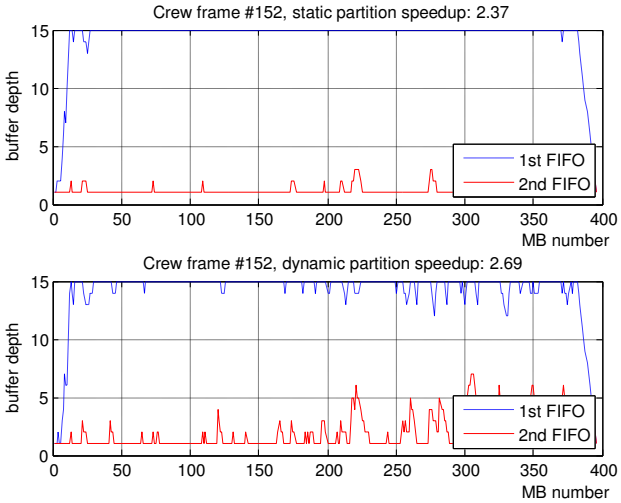

Fig. 15. The runtime buffer growth at frame #152 of the Crew@1.5 mbps sequence for the static and dynamic pipeline partition decoders.

MB prediction, and reconstruction. The third stage runs the in-loop filter. There are two circular FIFO buffers in the decoder. The first one is between stage one and two. The second one is between stage two and three. Note that each node in both the circular FIFO buffers has the same data structure and is 2K bytes in size. The speedup ratios depend directly on the underflow or overflow of the circular FIFO buffers. If the buffers are never underflow or overflow throughout the decoding of a video sequence, the average speedup ratio should be close to three. However, as the complexity of MB decoding varies significantly in a video frame, the buffers may underflow or overflow which reduces the decoding performance.

Fig. 15 shows the runtime buffer growth during the decoding of the frame #152 of the Crew@1.5 mbps sequence. Frame #152 has the lowest speedup ratio of 2.37 for the static partition pipeline decoder. Note that the sequence is in CIF resolution so there are 396 MBs per frame. From Fig. 15, one can see that the buffer depth of the first FIFO buffer grows above 15 nodes (30KB) after MB #26. Detail analysis shows that the

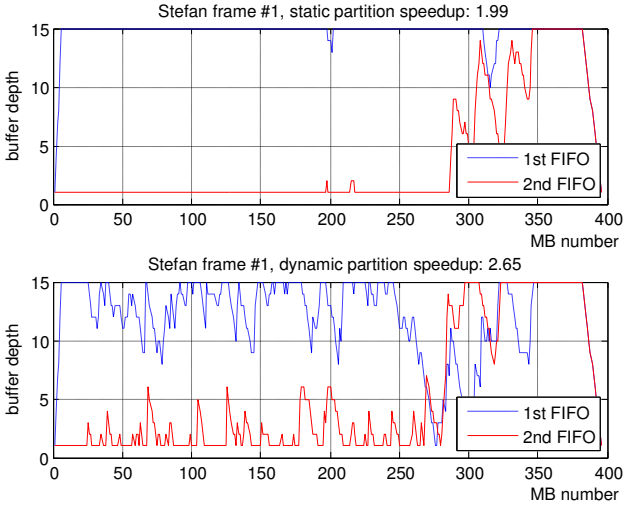

Fig. 16. The runtime buffer growth at frame #1 of the Stefan@512 kbps sequence for the static and dynamic pipeline partition decoders.

computation load of the second stage is too high such that the first FIFO is always full and the second FIFO contains only the MB node under processing. With dynamic stage partitioning, whenever there is a skip MB, its computation for the second stage will be offloaded to the first stage and the average speedup ratio increases to 2.67.

The skip MB-based dynamic pipeline partitioning policy achieves best performance gain with the Stefan sequence. Fig. 16 shows the runtime buffer depth growth during the decoding of frames #1 (i.e., the second frame in the video sequence) of the Stefan@512 kbps sequence. As one can see, the dynamic partitioning of pipeline stages effectively reduces the overflow of the first FIFO buffer and increases the speedup ratio by 33%.

VI. Experimental Results

In this section, we conduct some experiments to evaluate the performance of the proposed application processor architecture for video decoding. The video bitstreams used in this section are encoded using the AVC/H.264 JM reference software version 18.4 [29]. The baseline profile is used to encode five CIF (352×288) sequences, Crew, Foreman, Mobile, News, and Stefan, and an SD (720×480) sequence, Crew. All videos are 30 Hz sequences that have 300 frames. The motion estimation search range is set to 32. Finally, rate control is enabled to encode the CIF sequences at 512 kbps and 1.5 mbps, and the SD sequence at 1.5, 3.0, and 6.0 mbps.

The single-core AVC decoder extracted from the source code of the Google Android project [30] is used as the baseline reference decoder. We simply remove all the dependencies of the AVC decoder on the Android system and make it a stand-alone C application. There is no platform-dependent optimization done to the decoder source code. The proposed pipeline-based parallel decoder and the wavefront parallel decoder are both derived from the single-core reference decoder. The application program reads the whole bitstream data from the CF card into a large SDRAM buffer before the decoder starts. A cycle-accurate hardware timer is used to measure the performance of each decoder.

The four-core application processor with the proposed pipeline-based datapath is implemented on the Xilinx Vertex-5 FPGA platform, XUPV5-LX110T. The synthesizable RTL model of the IPC controller and the hardware mutex are written in the Verilog hardware description language. The rest of the IPs, including the Microblaze processor, the eight-port DDR2 DRAM controller, the on-chip memory controller, are soft-core IP's provided by the Xilinx EDA tools. The complete hardware-software system is integrated using the Xilinx XPS 13.4 and SDK 13.4.

The system clock rate is set to 83.3 MHz. The critical path of the system starts from the instruction decode unit of the Microblaze processor, passing through the IPC controller, and ends at the LMB connected to the on-chip BRAM (i.e., the scratchpad memory). The logic usages of the main IP's of the application processor are shown in TABLE II. Note that the IPC controller logic is quite small. Each Microblaze processor has 8KB instruction cache. The data cache size is set to 16KB for the proposed pipeline decoder and 64KB for the wavefront decoders. The local scratchpad memory for each core is 64KB for the pipeline decoder and 16KB for the wavefront decoder. Theoretically, the wavefront decoder does not need to use the local scratchpad memory. However, to compensate for potential loss due to the small 64KB cache size per core, we put the stack (i.e., local variables) of the wavefront decoder on the high-speed scratchpad memory so that the comparison against the proposed pipeline decoder would be more reasonable. The platform has 256MB of 64-bit DDR2 DRAM. There are some IPs used for profiling and debugging in the SoC that is not shown in Fig. 8, including a hardware timer, a hardware debug agent (Xilinx MDM IP), a RS232 UART controller, and the glue logic to the Xilinx SystemACE IC used for dumping the decoded video frames to the FAT32 file system on a Compact Flash (CF) card. The CF card file dumps are only used to verify the correctness of the decoder. For the decoder performance measurements, decoded frames are not saved. The hardware timer device is connected and shared to all processor cores via the Xilinx FSL buses. The timer device will neither issue interrupts nor generate system bus traffics. It is designed as a passive register that keeps system clock counts. All the time measurements in this section are readings from this timer register.

TABLE III shows the performance of the baseline single-core decoder on the target platform. All the time measurements are the average runtime of three trials of decoding the entire sequence. According to our experiments, the performance of the Microblaze core is about 1.2 DMIPS/MHz. It is not fast enough for real-time decoding of a 30Hz CIF video sequence encoded in AVC/H.264. However, our focus is on the speedup ratios of different multi-core parallel decoding schemes on the target platform.

TABLE IV shows the performance of the pipeline-based video decoder that uses the proposed IPC datapath for three-core parallel video decoding. Although the application

TABLE II
FPGA Logic Usage of the Proposed Application Processor

IP (single instance)	LUT6	Flip-Flops	RAM
Microblaze	2164	2391	8+16KB
IPC controller	567	297	
Local scratchpad memory			64KB
Hardware mutex	1103	568	
8-port memory controller	6580	7836	2KB

The FPGA device is a XC5VLX110T-FF1136. LUT6 is a 6-input FPGA lookup table. The system clock rate is 83.3 MHz.

TABLE III
Baseline Single-Core Video Decoder Performance

Video Sequences	512 kbps	1.5 mbps
Crew	97.57	136.55
Foreman	86.33	118.97
Mobile	85.61	116.87
News	57.15	81.27
Stefan	86.39	119.30

All sequences are in CIF resolution and have 300 frames. The numbers are decoding time in seconds. They are the average runtime of three trials. The test platform has 16KB of local scratchpad (not used here) and 64KB of data cache per processor core. Only one core is used to run the single-core video decoder.

processor we constructed has four Microblaze cores, the first core is exclusively used for system maintenance tasks including extracting NAL units from the bitstream in SDRAM, sending the NAL units to the decoding pipeline, and printing output messages to the RS232 device, etc. The remaining three cores are used exclusively to run the decoding pipeline. As one can see, the performance of the pipeline architecture is up to 2.89 times higher than that of the original single-core decoder. It would be interesting to verify how much of the performance gain is due to the proposed datapath. In TABLE V, the performance of a three-stage (i.e., three-core) pipeline decoder that uses shared buffers in DRAM, instead of the local scratchpad memory, to pass the macroblock decoding data between the pipeline stages. The synchronization mechanism and thread assignments of the DRAM-based pipeline decoder are the same as the proposed pipeline decoder. However, the three-core DRAM-based pipeline decoder is only up to 1.83 times faster than the single-core decoder. In short, the pipeline-based video decoder requires efficient IPC mechanism that cannot be provided by the traditional SMP architecture. The performance of the wavefront-based parallel video decoder is shown in TABLE VI. The implementation detail of the wavefront decoder is described in section III.*C*. The performance of the wavefront decoder is better than the performance the DRAM-based pipeline decoder but worse than that of the proposed pipeline decoder.

The speedup ratios of the proposed pipeline decoder, the DRAM-based pipeline decoder, and the wavefront decoder are summarized in TABLE VII and TABLE VIII. The speedup ratio is computed by dividing the decoding time of the single-core decoder by that of the three-core decoder. Theoretically, the speedup ratios would be close to three if the decoding algorithm scales perfectly. In practice, due to system bus or memory bandwidth issues and the IPC overheads, the speedup ratios are usually less than three. It is interesting to see that the speedup ratios usually increase for the pipeline decoders when the bitrate goes from 512 kbps to 1.5 mbps. With current dynamic pipeline stage partitioning scheme, higher video bitrate usually improves the load balance of the pipeline-based decoders.

On the other hand, the performance of the wavefront decoder actually drops as the bitrate increases. The reason is that the entropy decoding process of AVC/H.264 is sequential. As the bitrate becomes higher, it takes longer for the entropy decoder to finish decoding a frame and starts dispatching MB decoding tasks among all three cores. It is reasonable to try to interleave the entropy decoder and the MB decoding tasks so that the impact of the sequential entropy decoding process can be alleviated. However, it is not easy to distribute the entropy decoding tasks and all the MB decoding tasks evenly among all three processor cores. Therefore, we have implemented two interleaving wavefront video decoders, one with a three-core scheduling policy and one with a four-core scheduling policy on the target platform so that we can obtain some idea about the ideal speedup ratios of a three-core wavefront decoder on the target platform.

For the three-core interleaving wavefront video decoder, the first core performs entropy decoding row-by-row. Every time a row of MBs are entropy-decoded, the remaining part of its decoding task will be assigned to the second core or the third core. If both cores are busy with previous decoding tasks, the first core will enter the MB decoding loop by itself. For the four-core interleaving decoder, the first core is only responsible for entropy decoding. Every time one row of MBs are decoded, it will assign the rest of the MB decoding tasks of that row to the other three cores. TABLE IX shows the performance of the interleaving decoder. Surprisingly, at 512 kbps, the three-core interleaving wavefront decoder does not perform better than the wavefront decoder with the sequential entropy decoding module. At 1.5 mbps, the interleaving decoder only performs slightly better. The reason is probably due to the fact that the video decoding process has highly variable complexity. Thus, the three-core version does not interleave the entropy decoding tasks and the MB decoding tasks that well. The four-core interleaving wavefront decoder does perform much better than all other decoders (yet with one extra processor core). The speedup factors of the four-core interleaving decoder can be regarded as an upper-bound of the speedup factor of the three-core interleaving decoder on the target platform if good load balance can be achieved.

It is known that the performance of the wavefront decoder may scale better if the video resolution is larger. In TABLE X, we have tested the proposed three-core pipeline decoder and the three-core wavefront decoder using a 720×480 version of the Crew sequence encoded at three different bitrates: 1.5, 3.0, and 6.0 mbps. Again, the proposed pipeline decoder outperforms the wavefront decoder in all bitrates. What does not show in TABLE X is that the proposed pipeline decoder does not consume much extra intermediate data memory even if the video resolution is four times as large now. The reason is because that the pipeline-based video decoder reuses data structure for each macroblock at each stage without much dependency on the video resolution. On the other hand, the size of the intermediate data structure of the wavefront threads is often resolution dependent.

It is obvious that the scalability of a wavefront parallel video decoder heavily depends on the memory subsystem. We intentionally put the stack space (i.e., local variables of all functions) of the wavefront decoder on the high-speed on-chip scratchpad memory of the target FPGA platform to alleviate the performance influences of a smaller cache. To further justify the performance of our implementation of the wavefront decoder, we ran the decoder on the Nexus 7 2013 tablet computer. The tablet has a Qualcomm Snapdragon S4 Pro APQ8064–1AA processor with three levels of data caches (L0: 4KB, L1: 16KB, and L2:2MB). The compiler used to build the decoder is from the Android NDK r9b. TABLE XI shows the performance of the wavefront decoders on the Nexus 7 2013 tablet. From TABLE XI, it is clear that the wavefront decoder on the Nexus tablet and the FPGA platform have similar speedup ratios. In addition, in [5], a wavefront video decoder is implemented on an Intel X86 platform with four processor cores. Its maximal speedup ratio is about 2.7 when decoding test sequences similar to the ones used in this paper.

TABLE IV
PROPOSED THREE-CORE PIPELINE DECODER PERFORMANCE

Video Sequences	512 kbps	1.5 mbps
Crew	35.63	50.31
Foreman	30.91	41.17
Mobile	31.83	41.58
News	24.58	30.05
Stefan	33.50	45.35

All numbers are decoding time in seconds. The decoder uses the proposed pipeline datapath for decoding. Dynamic pipeline-stage partitioning is adopted. The test platform has 64KB of local scratchpad and 16KB of data cache per processor core.

TABLE V
DRAM-BASED THREE-CORE PIPELINE DECODER PERFORMANCE

Video Sequences	512 kbps	1.5 mbps
Crew	53.19	70.92
Foreman	47.81	61.47
Mobile	49.29	61.67
News	40.65	51.50
Stefan	50.68	64.63

All numbers are decoding time in seconds. The decoder uses the shared main memory (DDR2-DRAM) to store the pipeline FIFO buffers. The test platform has 16KB of local scratchpad (not used here) and 64KB of data cache per processor core.

TABLE VI
WAVEFRONT THREE-CORE DECODER PERFORMANCE

Video Sequences	512 kbps	1.5 mbps
Crew	42.86	63.00
Foreman	38.86	56.24
Mobile	38.82	56.14
News	29.59	44.80
Stefan	38.81	56.87

All numbers are decoding time in seconds. The test platform has 16KB of local scratchpad (not used here) and 64KB of data cache per processor core.

TABLE VII
SPEEDUP RATIO OF DIFFERENT DECODERS AT 512 KBPS

Sequences	Proposed Pipeline	DRAM Pipeline	Wavefront
Crew	2.74	1.83	2.28
Foreman	2.79	1.81	2.22
Mobile	2.69	1.74	2.21
News	2.33	1.41	1.93
Stefan	2.58	1.70	2.23

The single-core decoder is used as the reference decoder for the calculation of the speedup ratios of all the three-core decoders. Thus, the upper bound of the speedup factor is around 3.

TABLE VIII
SPEEDUP RATIO OF DIFFERENT DECODERS AT 1.5 MBPS

Sequences	Proposed Pipeline	DRAM Pipeline	Wavefront
Crew	2.71	1.93	2.17
Foreman	2.89	1.94	2.12
Mobile	2.81	1.90	2.08
News	2.70	1.58	1.81
Stefan	2.63	1.85	2.10

The single-core decoder is used as the reference decoder for the calculation of the speedup ratios of all the three-core decoders. Thus, the upper bound of the speedup factor is around 3.

TABLE IX
SPEEDUP RATIO OF WAVEFRONT WITH INTERLEAVED ENTROPY DECODER

Sequences	512 kbps		1.5 mbps	
	3-core	4-core	3-core	4-core
Crew	2.16	2.72	2.11	2.95
Foreman	2.11	2.61	2.05	2.83
Mobile	2.09	2.63	2.01	2.82
News	1.80	2.36	1.75	2.59
Stefan	2.07	2.62	1.99	2.80

TABLE X
SPEEDUP RATIO FOR A 720×480 VIDEO SEQUENCE

Bitrate	Proposed Pipeline Decoder	Wavefront Decoder
1.5 mbps	2.69	2.28
3.0 mbps	2.81	2.20
6.0 mbps	2.77	2.04

The single-core decoder is used as the reference decoder for the calculation of the speedup ratios of both three-core decoders. The decoding times of the single-core decoder are 314.7, 379.9, and 468.6 seconds, respectively. The video is the 300-frame Crew sequence.

TABLE XI
SPEEDUP RATIO OF WAVEFRONT DECODERS ON NEXUS 7 2013

Sequences	512 kbps		1.5 mbps	
	3-core (NIE)	4-core (IE)	3-core (NIE)	4-core (IE)
Crew	2.32	2.80	1.88	2.97
Foreman	2.20	2.65	1.89	2.81
Mobile	2.11	2.60	1.78	2.72
News	1.87	2.55	1.52	2.30
Stefan	2.16	2.62	1.85	2.75

The average decoding times of the single-core decoder for different videos are 6.44, 5.68, 5.66, 3.29, and 5.62 seconds, respectively.

The 3-core decoder does not interleave the entropy decoding (NIE) task with the MB decoding tasks while the 4-core decoder interleaves the entropy decoding (IE) task with the MB decoding tasks row-by-row. For the 4-core decoder, the first core is solely responsible for the entropy decoding task.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a multicore architecture with a dedicated pipeline datapath for the efficient execution of the data stream processing tasks. The pipeline datapath is a special hardwired IPC channel that allows each processor core to manipulate the data passing through it. The pipeline-like operate-and-transfer of a data item through the IPC channel does not consume any system bus bandwidth. It can be integrated into existing multicore application processors. Furthermore, it is easy to design software to take advantage of the proposed pipeline datapath. The proposed architecture with four processor cores has been implemented on a Xilinx Vertex-5 FPGA development board. We have also implemented two parallel video decoders on the platform to justify the benefits of the proposed architecture. The complete source code of the system, including the hardware model and the parallel decoder software, will be available in the public domain.

Although current implementation has shown promising results, some improvements should be made to the system. First, the data cache coherence across multiple cores is currently maintained by the application software. For general-purpose software pipeline researches, it is better to implement an MPSoC with a coherent cache. Secondly, the dynamic pipeline stage partitioning decision is only based on whether the

decoding mode is a skip MB or not. Although we have already seen 12% performance gain going from the static to the dynamic pipeline partitioning, more sophisticated algorithm is necessary to achieve optimal performance at runtime [3][9][12].

Currently, the thin OS kernel we have implemented for the proposed architecture does not support preemptive multitasking. In order to conduct experiments on concurrent executions of the traditional SMP applications and the proposed software pipeline applications, a more sophisticated OS kernel is required. We will try to port an RTOS to the proposed platform in the future. Finally, the current implementation of the IPC controller does not allow arbitrary assignment of a pipeline stage thread to a core. Such design simplifies the circuitry but put some constraint on the programming model. More flexible hardware support for the formation of the pipeline datapath will be investigated in the future.

REFERENCES

[1] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Trans. On Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1667, Dec. 2012.

[2] F. Bossen, "On Software Complexity: Decoding 1080p Content on a Smartphone," *Input document to 11th JCT-VC meeting, JCTVC-K0327*, Shanghai, China, 10-19, Oct. 2012.

[3] C.-J. Tsai, T.-F. Shen, P.-C. Liao, "Dynamic Task Partition for Video Decoding on Heterogeneous Dual-core Platforms," *ACM Trans. on Embedded Computing Systems*, **12**, 1s, Article 53, Mar., 2013.

[4] C. Meenderinck, A. Axevedo, B. Juurlink, M. Alvarez-Mesa, A. Ramirez, "Parallel Scalability of Video Decoders," *Journal of Signal Processing Systems for Signal, Image, and Video Technology*, Aug. 2008.

[5] S. H. Jo, S. Jo, and Y. H. Song, "Efficient Coordination of Parallel Threads of H.264/AVC Decoder for Performance Improvement," *IEEE Trans. on Consumer Electronics*, Vol. 56, No. 3, Aug. 2010, pp. 1963-1971.

[6] M. Alvarez-Mesa, C.C. Chi, B. Juurlink, V. George, V. and T. Schierl, "Parallel Video Decoding in the Emerging HEVC Standard," *Proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Kyoto, Japan, Mar. 2012, pp. 1545-1548.

[7] T.-C. Chen, C.-J. Lian, and L.-G. Chen, "Hardware Architecture Design of an H.264/AVC Video Codec," *Proc. of 11th Asia and South Pacific Design Automation Conf. (ASPDAC)*, Yokohama, Japan, 24-27, Jan. 2006, pp. 750-757.

[8] W.-J, Kim, K. Cho, and K.-S. Chung, "Stage-based Frame-Partitioned Parallelization of H.264/AVC Decoding," *IEEE Trans. on Consumer Electronics*, Vol. 56, No. 2, May 2010, pp. 1088-1096.

[9] C. Lee and S. Yang, "Design of an H.264 Decoder with Variable Pipeline and Smart Bus Arbiter," *Proc. of Int. SoC Design Conf. (ISOCC 2010)*, Seoul, Korea, 22-23, Nov., 2010.

[10] G. S. Tyson, M. Smelyanskiy, and E. S. Davidson, "Evaluating the Use of Register Queues in Software Pipelined Loops," *IEEE Trans. on Computers*, Vol. 50, No. 8, Aug. 2001, pp. 769-783.

[11] J. Wang and B. Su, "Software Pipelining Of Nested Loops For Real-Time Dsp Applications," *Proc. of the 1998 IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP'98)*, Seattle, USA, 12-15 May, 1998, pp. 3065-3068.

[12] J. Jahn, S. Pagani, S. Kobbe, J.-J. Chen, Joerg Henkel, "Optimizations for Configuring and Mapping Software Pipelines in Many Core Systems," *Proc. of 50th ACM/IEEE Design Automation Conf. (DAC'13)*, Austin, USA, May, 2013, pp. 1-8.

[13] B. Wang, S. Shang, Q. Fang, W. Zheng, "Parallel Task Developing Based on Software Pipeline in Multicore System," *Proc. of Int. Symp. on Parallel and Distributed Processing with Applications*, Taipei, Taiwan, 6-9, Sep., 2010, pp. 542-549.

[14] J. A. Pienaar, S. Chakradhar, and A. Raghunathan, "Automatic Generation of Software Pipelines for Heterogeneous Parallel Systems," *Proc. of Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, Utah, USA10-16 Nov., 2012, pp. 1-12.

[15] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," *Proc. of IEEE Int. Conf. on Compilers, Arch., and Synth., for Embedded Syst. (CASES)*, 2012, pp. 71-80.

[16] H. Lee, W. Che, and K. Chatha, "Dynamic Scheduling of Stream Programs on Embedded Multi-core Processors," *Proc. of Int. Symp. on Hardw./Softw. Codesign and Syst. Synth. (CODES+ISSS)*, 2012.

[17] D. F. Finchelstein, V. Sze, A. P. Chandrakasan, "Multicore Processing and Efficient On-Chip Caching for H.264 and Future Video Decoders," *IEEE Trans. on Circuits and Systems for Video Technology*, Vol. 19, No. 11, Nov., 2009, pp. 1704-1713.

[18] The Open Group, *The Open Group Base Specifications Issue 7, IEEE Std 1003.1™*, 2013.

[19] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC Video Coding Standard," *IEEE Trans. On Circuits and Systems for Video Technology*, Vol. 13, No. 7, July 2003, pp. 560–576.

[20] C. Lee, S. Yang, "Design of an H.264 Decoder with Variable Pipeline and Smart Bus Arbiter," *Proc. of ISOCC,* Seoul, Korea, 2010.

[21] T.-J. Lin et. al., "Overview of ITRI PAC Project – from VLIW DSP Processor to Multicore Computing Platform," *Proc. of IEEE VLSI-DAT'08*, Hsinchu, Taiwan, Apr., 2008.

[22] F. Pescador, G. Maturana, M.J. Garrido, and C. Sanz, "An H.264 Video Decoder based on a Latest Generation DSP, " *IEEE Trans. Consumer Electronics*, Vol. 55, No. 1, 2009, pp. 205–212.

[23] T. Ungerer, B. Robic, and J. Silc, "Multithreaded Processors," *The Computer Journal*, Vol. 45, No. 3, 2002, pp. 320-348.

[24] S.-W. Wang, S.-S. Yang, H.-M. Chen, C.-L. Yang, and J.-L. Wu, "A Multi-core Architecture based Parallel Framework for H.264/AVC Deblocking Filters," *Journal of Signal Processing Systems*, Dec. 2008.

[25] Y. Zhang, C. Yan, F. Dai, and Y. Ma, "Efficient Parallel Framework for H.264/AVC Deblocking Filter on Many-Core Platform," *IEEE Trans. on Multimedia*, Vol. 14, No. 3, June 2012, pp. 510-524.

[26] G. Khurana, A.A. Kassim, T. P. Chua, and M. B. Mi, "A Pipelined Hardware Implementation of In-loop Deblocking Filter in H.264/AVC," *IEEE Trans. on Consumer Electronics*, Vol. 52, No. 7, May 2006, pp.536-540.

[27] M. Kthiri, P. Kadionik, H. Lévi, H. Loukil, A. Ben Atitallah, and N. Masmoudi, "A Parallel Hardware Architecture of Deblocking Filter in H264/AVC," *Proc. 9th Int. Symp. on Electronics and Telecomm. (ISETC)*, Timisoara, Romania, 11-12 Nov.,2010, pp. 341-344.

[28] C. C. Chi and B. Juurlink, "A QHD-capable parallel H.264 decoder," *Proc. of the Int. Conf. on Supercomputing (ICS11)*, 2011, pp. 317-326.

[29] JM source available at http://iphome.hhi.de/suehring/tml/

[30] Android source available at http://source.android.com/

[31] G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115-116, 1981.

[32] K. Alagarsamy, "A mutual exclusion algorithm with optimally bounded bypasses," *Information Processing Letters*, vol. 96, no. 1, pp. 36-40, 2005.