Flexible DMA subsystem in multi-core platforms for video applications

Moonmo Koo and Soo-Ik Chae
School of EECS, Seoul National University.
Building 104–1, Seoul National University, Gwanakgu, Seoul, 151–742, Korea
a) jeffley@sdgroup.snu.ac.kr

Abstract: We propose a flexible DMA subsystem suitable for multi-core systems, in which DMA set-up routines are separated from DMA requesting threads and DMA completion flags can quickly be checked by DMA synchronizing threads. We will briefly describe its architecture and implementation. By using a multi-core DSP system with the proposed DMA subsystem, we implemented an H.264/AVC software decoder that can decode D1 30 frames per second when the system operating clock frequency is about 265 MHz, assuming that all cores are operated at the same system clock frequency. With experimental results for the H.264 decoder, we confirmed its flexibility and performance improvement.

Keywords: DMA, H.264 decoder, video application, multiprocessor, multi-core system

Classification: Science and engineering for electronics

References

1 Introduction

Because the multi-core platforms for high-performance video systems require massive data transfer, we employ a DMA subsystem that supports multiple outstanding DMA transfers to improve system performance and hide the latency of data transfer by overlapping CPU computation with the data transfer. Moreover, fast synchronization to the DMA transfers is very important for the performance of the video systems.

The operations related to a DMA request can be divided into three routines: set-up, data transfer, and synchronization. The set-up routine builds a DMA descriptor and issues it to a DMA engine. Address calculation for building the DMA descriptors is quite complex and irregular in the video decoders [2]. Therefore, we can enhance performance by separating the set-up routines from a DMA-requesting thread and gathering DMA descriptors to efficiently issue these to the DMA engine.

For DMA synchronization, it also takes a substantial number of clock cycles in the conventional DMA subsystems [5] for a synchronizing thread which waits a DMA transfer to be completed because it should first find a DMA channel allocated for the waiting DMA transfer and then check its completion flag. We can use one of the two basic methods for synchronization: either interrupt-based [1] or polling-based [3]. However, it is difficult to achieve fast synchronization with the interrupt-based method due to the relatively large context-switching overhead of interrupt-service routines particularly when frequent DMA transfers are requested. The polling-based method is neither suitable for multi-core platforms because of large overhead in checking the DMA completion flag in the DMA engine. However, this overhead can substantially be reduced by delivering the DMA completion flag in a local memory where the synchronizing threads can access quickly for polling.

In this letter, we propose a flexible DMA subsystem that off-loads DMA set-up operations from the DMA-requesting threads and employs a fast sync scheme which delivers each DMA completion flag near to its DMA-synchronizing threads. In Section 2, the architecture of the proposed DMA system and its implementation are described in detail. Experimental results for an H.264 baseline D1 decoder are presented in Section 3, which is followed by a brief conclusion.

2 Proposed DMA subsystem

We propose a DMA subsystem for a multi-core system, which is composed of one or more DMA engines and one or more DMA managers, as illustrated in Fig. 1 (a). The DMA engine is a hardware component with multiple channels, which is not directly accessed by the application threads. Each DMA channel executes a data transfer specified with a source address, a destination address, and a transfer size. And a DMA manager consists of three types of functional blocks such as descriptor generators, request dispatchers, and sync flag dispatchers. The proposed DMA subsystem is easily portable
because all the threads in the DMA manager are implemented in software. Therefore, the descriptor generators and sync flag dispatchers are fully configurable according to the DMA requirements of an application. Similarly, the request dispatchers are easily adaptable for the various hardware DMA engines, especially when the number of DMA channels is modified.

As shown in Fig. 1 (a), each component in the DMA manager represents a processing step for DMA requests. A requesting thread requests a DMA transfer by sending a set of the parameters $A$ to its descriptor generator, which builds DMA descriptors with the given parameters. In H.264/AVC, these parameters can be a macroblock (MB) index, a motion vector, a prediction mode, etc. [2] and a DMA descriptor consists of source and destination addresses, a transfer size, and its operation mode. Each descriptor generator puts a descriptor $B$ to its corresponding request queue for a request dispatcher, which finds an available channel in the DMA engines and selects a request in one of the request queues before issuing the selected request $C$.

Each synchronizing thread, which waits for a DMA transfer, prepares and resets, when it is activated, a sync flag for the waiting DMA transfer. The sync flag dispatcher checks the completion flag for each DMA transfer from its corresponding DMA channel. If the DMA engine finds a completed transfer, it sends the completion flag $D$ to the sync flag dispatcher, which should distribute it as the sync flag $E$ to the local memories of all synchronizing threads waiting for the completed DMA transfer so that they can quickly access the sync flag.

A CPU controls a DMA engine [5] through an AMBA bus in a single-core system where the roles of the descriptor generators, the request dispatcher, and the sync flag dispatcher should be carried out by either application or OS threads in the same CPU. In a multi-core system, however, they can be off-loaded to other processor cores for high-performance applications. Furthermore, the sync flag dispatcher reduces the bus contention substantially by delivering the sync flag to a local memory close to each synchronizing thread. It is also preferable that the components in the DMA manager can
be distributed over several processors to accelerate the DMA operations.

We implemented the proposed DMA subsystem for a bus-based multi-core system with multiple banks of shared memory, as shown in Fig. 1 (b). We employed a communication link between two components as a FIFO queue. Implementation of the queues can be varied according to spatial and temporal distributions of requesting and synchronizing threads. To reduce the number of the queues, queue sharing is employed although it increases the latency. Therefore, performance and complexity can be trade off.

The numbers from (1) to (9) in Fig. 1 (b) indicate the steps for processing a DMA transfer. (1) A thread in a processor writes the parameters of a DMA request in the parameter buffer. (2) The processor, which is a bus master, sends the identifier (ID) of the outstanding DMA request to the request collector, which is a bus slave. (3) The request collector puts the ID to a shared queue if the queue is not full. (4) The descriptor generator gets a request ID from the queue, if not empty. (5) Then it gets the parameters with the ID from the parameter buffer to make a DMA descriptor, which is passed to the request dispatcher. (6) The request dispatcher first finds an available channel in the DMA engine and then issues the DMA descriptor to the DMA channel. (7) After completing a DMA transfer, the DMA engine sends its completion flag to a completion flag buffer. (8) The sync flag dispatcher sets a sync flag based on the updated completion flag buffer and sends it to its corresponding sync flag buffer. (9) Each synchronizing thread reads its sync flag buffer.

Assume that $T_R$, $T_T$, $T_S$, and $T_{SD}$ for a DMA transaction represent the time interval from its request to DMA command issue (step 1 to step 6), that for DMA data transfer, that from completing DMA data transfer to setting the sync flag (step 7 and step 8), and that from the DMA request to its synchronizing time point. The latency $T_R$ can be reduced by employing more than one request FIFO or more than one DMA manager thread, which shortens the queuing delay of the requests in the request queue. The waiting time of a thread for completing a DMA transfer can be represented as max $(T_R + T_T + T_S) - T_{SD}, 0)$. Therefore, no performance degradation occurs if $T_{SD} \geq (T_R + T_T + T_S)$ for all synchronizing threads.

3 Experimental results

For experiments, we used a multi-processor system that consists of three two-issue DSPs, one RISC processor, three DMA engines and a 32KB shared local memory with 6 banks. Each of the three DSPs supports specific video instructions for variable length decoding (DSP_VLD), motion compensation (DSP_MC), and integer transformation and deblocking filter (DSP_ITQ_DF), respectively.

We first implemented an H.264 D1 30 fps software decoder without employing the proposed DMA subsystem on the multi-processor system. We collected its performance data by running Foreman sequence on its FPGA implementation with the operating frequency of 25 MHz shown in Fig. 2 (b).
Function blocks in the H.264 decoder are executed in parallel while pipelined in the macroblock level, among which variable length decoding (VLD) block is the slowest one. The software decoder requires 412.1 M cycles to decode D1 30 frames. We reduced communication overhead by employing inter-process communication accelerators each of which consists of a thread scheduler, a context switch controller, and so on, which are not covered in this letter.

We also implemented another H.264 D1 30 fps decoder using the proposed DMA subsystem with two DMA managers and a fast sync scheme, which requires about 265 M cycles for decoding D1 30 frames, as summarized in Table I. In the DMA subsystem, the parameter buffer and the sync flag buffer are allocated into the shared local memory in Fig. 2 (a) while the completion buffer is replaced with the sync flag buffer to reduce $T_S$. Furthermore, a FIFO queue controller manages multiple DMA request queues each of which is dedicated to a DMA manager. One DMA manager is for motion compensation and the other one for variable length decoding, integer transform, and deblocking filtering. The former is mapped to DSP_MC while the latter to the RISC processor.

Table I. Performance and $T_R$ variation according to the number of DMA managers and the sync scheme

<table>
<thead>
<tr>
<th>$N$: # of DMA managers</th>
<th>Fast sync scheme</th>
<th>$T_R$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Off</td>
<td>On</td>
</tr>
<tr>
<td>$N = 1$</td>
<td>290.3 M</td>
<td>288.5 M</td>
</tr>
<tr>
<td>$N = 2$</td>
<td>274.2 M</td>
<td>264.9 M</td>
</tr>
<tr>
<td>$N = 3$</td>
<td>293.1 M</td>
<td>271.7 M</td>
</tr>
</tbody>
</table>

The speedup of the decoder is obtained from off-loading of DMA set-up routines and using fast synchronization for the DMA transfers in the VLD block which is the bottleneck of the performance. Computation load of the DMA setup routines that are offloaded to the RISC processor is 159.3 M cycles per second, and the fast sync scheme leads the cycle reduction by 12.9 M cycles per second. However, the waiting time of the VLD thread
for DMA completion is increased by 25.0 M cycles per second because off-loading DMA setup routines substantially reduces $T_{SD}$. Consequently, the clock cycles required to decode D1 30 frames are reduced by 147.2 M cycles to 264.9 M cycles. We also have confirmed that the multi-core system can be synthesized at an operating frequency of 350 MHz using the worst case device parameters of a 65 nm low-power ASIC library, which implies that the decoder can decode up to D1 39.6 frames per second.

Table I summarizes the performance enhancement of the decoder with the fast sync scheme and the $T_R$ variation when the number of DMA managers varies. According to Table I, $T_R$ is decreased as the number of DMA managers is increased. $T_R$ for a DMA request is the sum of its waiting time in a DMA queue, the time for a DMA manager to get it from the queue, and the time for the DMA manager to handle it. The waiting time in the queue can substantially be reduced by using multiple DMA managers each of which can handle a DMA request. In other words, each thread issued a DMA request needs to wait longer for its completion when we employ fewer DMA managers. Therefore, performance enhancement with the fast sync scheme gets smaller as the number of DMA managers is decreased, as shown in Table I. Note that the decoder performance is degraded a little with three DMA managers as shown in Table I. It is because the third DMA manager for integer transform and deblocking filter is mapped to DSP_ITQ_DF so that it shares the computation cycles of DSP_ITQ_DF with its pre-existing threads.

To achieve the performance for HD image decoding, we should further reduce by using the following schemes: (1) distributing the task of the DMA manager into more processors by increasing the number of its threads, (2) employing hardware acceleration for the DMA manager threads, and (3) reducing the number of clock cycles required to access the request FIFO, e.g. by implementing special instructions for the FIFO access. Additionally, we should also decrease $T_{FP}$ by the following schemes: (1) adopting a high-bandwidth DRAM and an efficient SDRAM controller [4], (2) using a wider memory data bus, and (3) linking multiple DMA transfers for inter-prediction of a macroblock where each DMA transfer is arranged to deliver the minimum number of pixels for its corresponding sub-block of the macroblock in H.264 [6]. Using the scheme (3) can reduce the required memory bandwidth substantially because the data transfer for inter-prediction occupies 73.4% of the total bandwidth of DMA data transfers, which corresponds to 59.6% of the total bandwidth to the SDRAM.

### 4 Conclusion and future work

In this letter, we proposed a flexible DMA subsystem suitable for multi-core systems that off-loads DMA set-up routines and employs a fast sync scheme. And we explained its architecture for flexibility. Moreover, we implemented an H.264 software decoder with the proposed DMA subsystem mapped to a multiple DSP system.
As a future work, we plan to explore the design space by changing the number of DMA engines and the number of DMA channels, and then implement an HD software video decoder by using a multi-core platform with the proposed DMA subsystem.

Acknowledgments

This work was supported by the Seoul R&BD Program (10560), and ISRC in Seoul National University.