An Analysis of Blocking vs Non-Blocking Flow Control in On-Chip Networks

ABSTRACT

High end System-on-Chip (SoC) architectures consist of tens of processing engines. These processing engines have varied traffic profiles consisting of priority traffic that require that the latency of the traffic is minimized, controlled bandwidth traffic that require low service jitter on the throughput, and best effort traffic that can tolerate highly variable service. In this paper, we investigate the trade-off between multi-threaded non-blocking (MTNB) flow-control and single threaded tag (STT) based flow-control in the realm of Open Core Protocol (OCP) [1] specifications. Specifically, we argue that the non-blocking multi-threaded flow-control protocol is more suitable for latency minimization of the priority traffic and jitter minimization of controlled bandwidth traffic, when compared with a single threaded tag (STT) based protocol. We present experimental results comparing MTNB against STT based protocols on representative DTV data flows. On average, in the STT based system, the latency of priority traffic is increased by 2.73 times and the latency of controlled bandwidth traffic is increased by 1.14 times when compared to the MTNB system, under identical configurations.

1. Introduction

A System-on-Chip (SoC) consists of several processing engines integrated onto the same chip. The traffic flow from each of the processing engines can have different performance requirements. For example, a CPU whose traffic to the external DRAM is dominated by cache-line fills would require that the latency of the traffic is minimized. On the other hand, traffic flow from a video decoder engine requires that the traffic is serviced with low jitter, such that the amount of buffering inside the engine can be minimized while ensuring that the engine never starves for data.

Figure 1: A typical video SoC

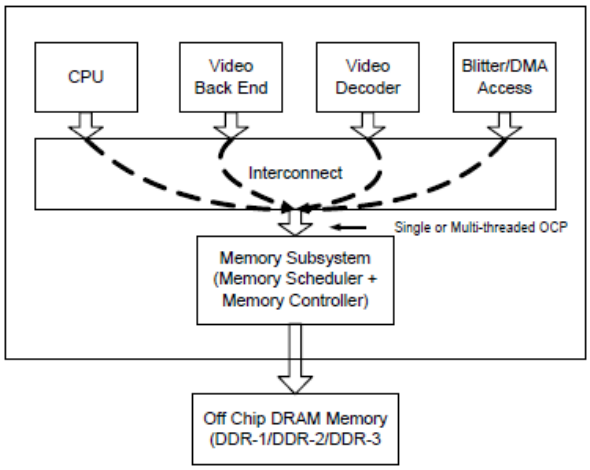


Figure 1 depicts a typical SoC architecture for HDTV systems [2]. For clarity, only the major blocks are included. The traffic flows from the different processing engines (henceforth called Initiators) converge into the DRAM memory. The memory subsystem consists of a memory scheduler, a memory controller, and the off-chip DRAM. The memory scheduler arbitrates among multiple traffic flows to generate requests to the controller, which are thenissued to the DRAM. Cost and performance requirements of SoCs dictate that the DRAM system operate at high transfer efficiency

while minimizing latency for high priority flows and guaranteeing throughput for real-time flow. SoC memory schedulers rely upon transaction re-ordering to optimize these tradeoffs. OCP provides two different out-of-order transaction models: a single threaded tag (STT) based system, or a multi-threaded nonblocking flow-control (MTNB) based system. In the STT system, a single queue is used, in which the insertion is in-order, and de-queuing operation can be out of order. The memory scheduler looks into the queue and re-orders the requests such that the performance goals are met.

The MTNB system uses threads, which are similar to virtual channels [3]. A MTNB memory subsystem manages per-thread queues. The memory scheduler looks at the head of the queue of each thread, and schedules a request to the memory controller from one of the threads. Therefore, it maintains the order within each thread, while interleaving the requests from different threads.

The choice of implementing an STT based flow-control protocol or an MTNB based flow-control protocol is driven by performance requirements and area minimization goals within the SoC. Proponents of STT based flow-control protocol argue that it is efficient because transaction storage is shared across all initiators and all pending requests are candidates for re-ordering to optimize performance.

An MTNB based protocol allocates separate queues for each traffic flow. Therefore, the designer can optimize the queue depths based on the distribution of traffic among the threads. Moreover, the MTNB system isolates the performance characteristics of threads from each other. An MTNB scheduler can back-pressure the traffic on over burdened threads to ensure that well behaved threads receive their allocated service.

In contrast to the MTNB based protocol, the STT protocol is highly sensitive to the traffic shaping characteristics at the initiators and the interconnection network. The memory scheduler has only one queue, and has to backpressure the interconnection network whenever this queue is filled. Thus, the scheduler has no control over the type of traffic arriving into its queues. Hence, all the entries in the queue may be occupied by a collection of bursty best effort traffic, causing the scheduler to back-pressure the network, thus preventing higher priority traffic from reaching the scheduler. Therefore, the STT based scheduler can only reliably achieve the required system performance when the traffic from the different initiators is carefully shaped – both within an initiator and across entire service classes. Conversely, the achievable performance for the latency and bandwidth sensitive data flows can be severely impacted when the traffic is composed of heterogeneous flows with diverse performance requirements.

DRAM chips used in high-end SoCs operate at 533 MHz (with data pin rates twice that) or higher. The memory scheduler should normally operate at similar frequencies to deliver acceptable throughput. In a MTNB system, the number of threads defines the fan-in to the scheduler, and thus helps determine the achievable frequency. The system-level SoC design challenge includes optimal mapping of the traffic flows to a smaller number of scheduler threads to obtain the desired performance-area-timing trade-off. Mapping algorithms normally allocate flows with similar characteristics and constraints to the same threads, and allocate relatively fewer flows per low latency or controlled bandwidth thread than per best-effort thread. Such a mapping will minimize the number of threads and thus help in achieving higher operating frequency in the scheduler and minimize area overhead.

In the STT system, the depth of the single queue defines the fan-in to the scheduler. This depth is typically much higher than the number of threads in the MTNB system so the STT scheduler can minimize the frequency of full queues back-pressuring into the network. Since the SoC designer does not have the flexibility to optimize the size of the queue per initiator, the single queue size must satisfy all operating scenarios. This can force the designer to assume

worst-case conditions and over estimate the size of the queue. Since the STT scheduler examines all the entries in the queue to make its scheduling decision, the increased queue depth increases the complexity of the scheduler and results in lower operating frequencies or deeper, less efficient scheduling pipelines than the MTNB system.

The remaining part of the paper is organized as follows: In Section 2, we describe related work. In Section 3, we provide an overview of thread versus tag vis-à-vis the OCP protocol. In Section 4, we provide details of the memory subsystem. In Section 5, we describe the STT protocol in detail. In Section 6, we describe the MTNB protocol in detail. In Section 7, we present experimental results, and finally, in Section 8, we conclude the paper.

2. Previous Work

In the past, researchers have spent considerable effort attempting to achieve performance guarantees in a NoC system. Goossens et al. [4] presented the Aethereal network-on-chip that provides guaranteed throughput and bounded latency. Vellanki et al. [5] presented a NoC architecture where the traffic profile is divided into two categories: best effort, and guaranteed throughput, and the NoC assigns higher priority to the guaranteed throughput traffic over the best effort traffic. Weber et al. [6] presented an analytical analysis to achieve bounded latency and service jitter of a heterogeneous traffic system. Marescaux et al. [7] presented an architecture called SuperGT, where time slots are reserved for guaranteed throughput and best effort traffic respectively, to achieve the required performance guarantees. Bolotin et al. [8] and Srinivasan et al. [9] address the latency minimization problem at network design time by routing latency sensitive traffic through minimum number of router hops. All these papers apply QoS at the interconnection network and do not take its effect on the memory subsystem into account. For example, in order to service latency sensitive traffic, the interconnection may inject traffic into the memory subsystem that is detrimental to DRAM efficiency (due to high page miss rate and read to write direction turnaround). In contrast, our work in this paper uses a scheme that closely follows the technique presented in [6], and applies it at the memory subsystem. Thus, it is able to gracefully trade-off the QoS requirements of the latency sensitive traffic, while maintaining high memory utilization.

The attempt to standardize on-chip communication interfaces has led to the advent of protocols such as OCP and AMBA AXI [10]. While OCP supports both MTNB and STT based protocols, AXI supports only STT based protocol. To the best of our knowledge, there is a dearth of qualitative and quantitative analysis to determine the relative advantages and disadvantages of the two protocols. In fact, going by prior published work, there seems to be a general lack of clarity on MTNB and STT based protocols and their implications on the performance of the SoC. For example, adopting an AMBA protocol automatically forces the communication interfaces to be STT based, and the initiators threads must be collapsed into a single target thread. However, adopting an OCP protocol may allow the initiators to be routed along individual threads. These system-level decisions play a critical role in the optimization of the NoC for the desired power, area and performance. The primary focus of our work is to describe the two protocols and then analyze them qualitatively and quantitatively in a real video SoC environment.

3. OCP Tags versus OCP Threads

The Open Core Protocol (OCP) [1] is a point-to-point communication standard developed for interfacing between on-chip components such as initiators, communication networks, and memory schedulers. The protocol provides a layered set of communication semantics that define threads, tags, transactions, transfer phases and flow control. It also provides a high degree of configurability to choose a subset of the full semantics that best meets the needs of a specific interface. The highest layer in OCP is the thread, which defines sequences of transactions that are unordered with respect to transactions on other threads. OCP supports optional per-thread flow control, enabling the delivery of fully non-blocking

interfaces in a manner analogous to virtual channels with virtual flit flow control [3]. In a MTNB interface, the receiver of each transfer phase implements a FIFO per thread, guaranteeing that the interface cannot block by asserting flow control on those threads with a full FIFO. Within a thread, transactions may have independent ordering tags; transactions with the same tag (including the common degenerate case of interfaces supporting the null tag) are ordered. The ordering restrictions apply to both transaction completion at the target (ensuring memory consistency) and response ordering at the initiator. The OCP tag mechanism is similar in principle to the ID mechanism proposed in the AMBA AXI system architecture [10]. Since tagged transactions have shared flow control, a shared queue model is implied where re-ordering is typically only practiced at the target, since deadlock-free re-ordering in the network would require storage for complete transactions. Tag-based systems should thus minimize transaction lengths to prevent a lower priority transaction from blocking a later, higher priority request from reaching the target. While both threads and tags can coexist on the same OCP interface, existing schedulers implement either the MTNB (multiple threads, but only a null tag per thread) or the STT (a single thread with multiple tags) approaches to minimize complexity.

4. Details of Memory Subsystem

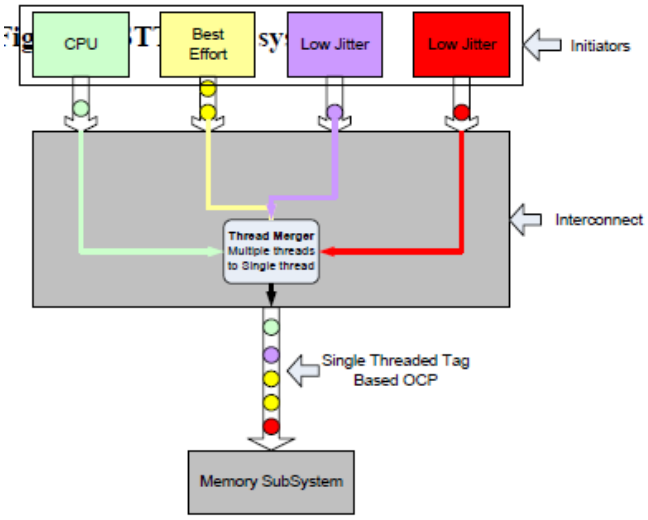
The memory subsystem in a SoC consists of a memory scheduler, memory controller, and the DRAM subsystem. The memory scheduler re-orders requests coming in from the interconnection network, such that a graceful trade-off can be obtained between memory efficiency and QoS requirements of latency sensitive traffic.

In this paper, we have modeled two memory schedulers: i) A thread based scheduler, and ii) a tag based scheduler. Both schedulers apply the same optimization functions to issue requests to the memory. The only difference is that the tag based scheduler uses a single thread and re-orders requests within the thread based on tag ID, while the thread based system looks at the head of the threads picks a winner from one of the threads to issue to the memory.

The memory scheduler applies several optimization functions before choosing the request to be sent to the memory. In order to improve efficiency, it chooses requests that target the open DRAM pages, requests that do not change direction from read to write and vice versa, and requests that are targeted to the same rank in the DRAM.

The memory scheduler applies a sophisticated QoS mechanism to ensure that the QoS requirements of the threads (tags) are met. It allows the user to specify three levels of QoS per thread (tag): Priority,Controlled Bandwidth, and Best Effort. A priority thread gets highest preference, followed by controlled bandwidth thread, which is the low jitter traffic, and finally, the best effort thread, where the traffic is not expected to have any QoS requirement. Similar to the technique presented in [6], the memory scheduler maintains credit counters and dynamically varies the QoS levels of the threads at run-time to ensure that all threads are serviced within their respective bandwidth requirement. At any cycle, the memory scheduler prefers requests that are in the elevated priority mode, followed by those in the elevated controlled bandwidth mode, followed by the best effort mode. Among the threads in the same mode, the scheduler prefers requests that results in maximum memory utilization.

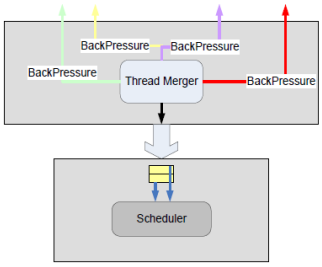
Figure 2



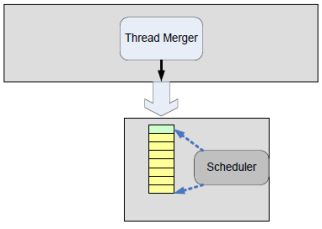
5. STT Based Flow Control Protocol

Figure 2 depicts a STT based flow control protocol. Note that any OCP interface can apply this protocol.

Figure 3: Shallow versus deep queue depth



The memory scheduler tries to optimize for area by reducing the buffering inside the scheduler. As a result, priority thread (green) is getting backpressured due to best effort thread (yellow)



The memory scheduler tries to increase its buffering to avoid backpressuring priority traffic (green). Now, the scheduler gets complex as it has to look into several entries in the queue, which will affect its timing closure

For simplicity, we have assumed that all the initiator OCP interfaces are singly threaded and tagged. In other words, each initiator generates streams of ordered OCP transactions, but each initiator manages its own thread. The interconnection network merges the separate initiator threads into a single threaded OCP interface connected to the memory subsystem, mapping the independent threads into an independent tag per initiator. The initiator tag enables re-ordering while indicating data flow service requirements to enable the QoS mechanisms of the scheduler. In the example, we assume that the traffic from the CPU traffic flow requires low average latency. The controlled bandwidth traffic should guarantee that its bandwidth is serviced with minimum area for buffering, and the best effort traffic can use high throughput when available, but does not have realtime requirements.

The memory scheduler examines all the entries in the queue, and makes a scheduling decision every cycle. As mentioned before, the scheduler prefers priority and bandwidth allocated traffic over best effort traffic. Because read-type transactions require much more response than request bandwidth and DRAM protocols loose efficiency due to bank refresh, readwrite data bus turnarounds and other factors, memory scheduler request queues often fill up and backpressure the interconnection network, thus preventing the network from sending further requests to the scheduler.

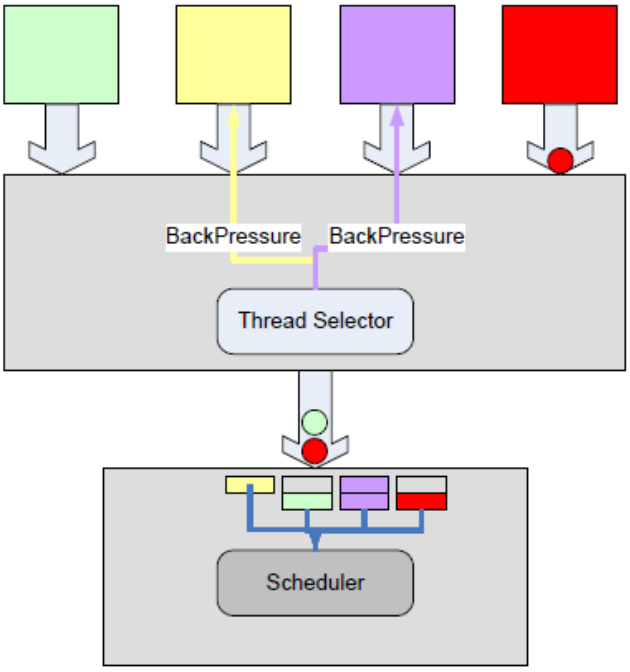
A couple of issues become apparent from the example shown in Figure 3. In the left hand side part of the figure, the designer optimizes for area, and chooses a small queue depth. We notice that the queue fills up quickly, and all the entries in the queue are from best-effort threads. This results in the priority traffic being back-pressured while the best-effort traffic is being serviced. In the right hand side part of the figure, the designer increases the size of the queue to recover the performance loss. Assuming that the queue size is 16 entries, the scheduler has to consider the inputs from each of the 16 entries to determine the request to be forwarded to the DRAM. A fan-in of 16 considerably reduces the maximum achievable frequency for the scheduler. Furthermore, the required queue depth required to protect the priority traffic grows with both the number of initiators and the count of their transactions that may be simultaneously active; both values increase as Moore's Law enables more complex SoCs and faster DRAMs require deeper memory pipelines.

6. MTNB Based Flow Control

Figure 4 depicts the MTNB based flow control protocol. Again, we assume that all the initiators are single threaded. However, the initiators are each mapped to individual threads at the OCP interface

connected to the memory subsystem. We assume the same characteristics for the traffic flows of the initiators as in the STT case.

Figure 4: MTNB Protocol



The memory scheduler has separate queues for each thread, hence, if the best effort thread's queue fills up, it does not back pressure other threads. As a result, more latency sensitive traffic can be serviced

Since there are individual buffers associated with each thread, it gives the user an opportunity to optimize the queue depth of each thread based on the volume of traffic. In the figure, the queues associated with the yellow (controlled bandwidth) and red (best Long bur effort) initiators are full, so their traffic is back-pressured, but the other initiators are not blocked. Comparing the MTNB and the STT set-up, assume the total depth of the queues is the same. In other words, $\Sigma D_{M,i} = D_S$, where $D_{M,i}$ is the queue depth corresponding to thread T_i in the MTNB system, and D_S is the queue depth of the STT system. The MTNB can service the priority traffic much better, as it is not back-pressured due to the best-effort traffic. Also, the scheduler complexity is much lower, as the scheduler only has a fan-in from the head of each thread, instead of the entire queue.

7. Experimental Results

In this section, we present results comparing the MTNB and the STT mechanisms for a representative DTV application. Further, we created two synthetic application benchmarks by modifying the DTV application to increase the number of router hops and increasing the bandwidth on the best effort traffic, respectively. The results section is organized as follows: In Section 7.1, we describe the essential performance measures that are needed to evaluate the performance of the system. In Section 7.2, we describe the application benchmarks. In Section 7.3, we describe the experimental setup, including the description about the benchmarks, and in Section 7.4, we present the summary of results.

7.1 Performance Evaluation Measures

The system-level performance of the SoC consists of bandwidth measurement at DRAM memory, latency measurement for the priority traffic, and the jitter measurement for the controlled bandwidth traffic. The bandwidth measurement at the DRAM memory is described in Mbytes per second (MBps).

The traffic from the priority thread consists of mainly CPU cache line misses and interrupts. For the cache line misses, one would be interested in the average latency for the traffic. On the other hand, for the interrupt traffic, the most important measurement is the worst case latency. Therefore, for the priority traffic, we measure the average latency as well as the worst case latency.

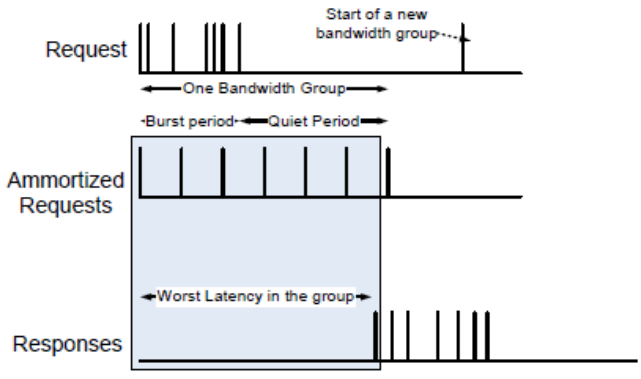
The controlled bandwidth traffic operates as follows: In the burst mode, the processing engine does some processing, and fills up its buffers with requests, which could be reads or writes. Once the burst mode comes to an end, the processing engine goes to a quiet mode, during which it does not do any processing. We denote one combination of a burst and quiet mode as a bandwidth group. The burst and quiet modes alternate such that the average bandwidth over a burst mode and a quite mode is the requested bandwidth at the initiator.

In the burst mode, as long as there is space in the buffers, the processing engine can continue to fill the buffers. When the buffers are full, the processing engine stalls and waits for a slot to open up in the buffer to start processing again. A slot opens up in the buffer on completion of a transaction.

Table 1: Traffic Profiles

Initiator	Fetch Unit	Typical burst size (Bytes)	Apprx %BW
ARM1176/ MIPS 74K	8-word cache line	32 bytes	15%
Video-Decoder (H.264 Macrobloc k)	16R x 16C x 16b per pixel	32 bytes per row	25%
Video Back End	Long bursts	256 bytes	40%
Blitter	Long bursts	256 bytes	15%
Low activity initiators	Short bursts	<32 bytes	5%

Figure 5: Jitter Measurement



The processing engine can choose to have a deep buffer to make sure that it never stalls. However this incurs additional gate costs. Therefore, the goal is to minimize the buffer size in the engine, and at the same time, ensure that the processing engine never stalls.

An estimation of the buffering required in the processing engine can be obtained from the worst latency experienced in one bandwidth group. We note that the traffic is requested as a burst followed by a quiet period, which can be amortized into a traffic that is uniformly distributed, with the specified bandwidth.

For illustration, refer to Figure 5. For this example, the latency of the first burst is the maximum, which results in the PE having to store 6 requests. Therefore, the smaller the "worst" latency within a bandwidth group the smaller the buffering required. The minimum amount of buffering required is given by the worst latency among all the bandwidth groups, among all bandwidth groups.

7.2 Application Benchmarks

In order to compare the performance of STT and MTNB protocols, we performed experiments with a representative DTV application, and two synthetic derivatives of the same.

DTV Application:

We compare the STT and MTNB protocols by applying them to a representative DTV benchmark that consists of seven initiators communicating through an external DRAM memory. The initiators are composed of a CPU, a video decoder, a video back-end, a transport engine, an audio processing unit, a blitter, and one initiator modeling other low level traffic. The traffic distribution of the DTV benchmark is described in Table 1.

The application had a read-to-write ratio of about 2:1. In the rest of the paper, we denote this application as DTV. The SoC architecture of the system is similar to the one depicted in Figure 1, except that there are seven initiators. In the MTNB system, each initiator sends traffic on its dedicated thread, which is mapped to a unique thread at the target. Therefore, the memory scheduler operates on seven threads. In the STT system as well, each initiator sends traffic on its dedicated thread. However, all initiator threads are collapsed to a single target thread. The traffic originating from each initiator thread has a unique tag ID, given by the initiator's ID. The memory scheduler

can look into its queue and re-order requests belonging to different tag IDs to improve performance.

For the MTNB system, we optimized the per thread queue depth required in the memory subsystem, such that traffic flows with higher bandwidth requirements were given deeper queues, while those with low bandwidth requirements were assigned shallow queues. The total number of buffers was calculated as the sum of the number of threads times the depth of the queue per thread. We used total buffers of 8 and 16 respectively, in our comparison. Therefore, we have assumed the same area overhead in terms of buffers for both MTNB and STT system.

Synthetic Applications:

For the synthetic application, our aim was to compare the performance of STT and MTNB based systems when: i) the system is stressed by a best-effort thread and, ii) the effect of increasing the number of pipeline stages (number of router hops from the initiator to the memory) on the performance of the system. The former case models the implications of increasing numbers of medium to high bandwidth initiators, such as I/O and graphics acceleration, in SoCs. The latter case considers deeper networks due to both higher total initiator count and higher operating frequencies.

We created the first synthetic application by increasing the bandwidth requirement at one best effort initiator such that it injects a transaction at every cycle. We denote this application as DTVstress. We created the second synthetic application by introducing one extra router hop between the initiators and the DRAM memory. We denote this system as DTVpipelined.

7.3 Experimental Setup

We utilized interconnect and memory subsystem architectures provided to us by Sonics, Inc [11]. We used SystemC models of the SonicsMX (SMX) interconnect along with the MemMax memory scheduler, and the MemDDRC DRAM controller plus external DRAM model. The interface between SMX and MemMax is naturally MTNB; in order to support the STT protocol, we modified the MemMax scheduler to model a single threaded interface with multiple tags.

7.4 Results

In this section, we present results for the different benchmarks presented above. Figures 6 through Figure 11 summarize the results. In each figure, the bar on the left denotes the MTNB system, and the bar on the right denotes the STT system. The values in the figure are normalized to the MTNB system. The X-axis denotes the four measurements of interests: i) Worst case latency of the CPU, ii) Average latency of the CPU, iii) Worst case latency per bandwidth group of the controlled bandwidth initiator and iv) Bandwidth serviced at the memory interface.

Figure 6 depicts the result comparing the MTNB and STT systems for the DTV benchmark, with the queue depth set to 8. Figure 7 depicts the corresponding results for the DTV benchmark with a buffer size set to 16. With queue depth set at 8, the STT system results in 3.3 times the worst case latency and 2.2 times the average latency for the CPU, and 0.7 times the bandwidth group latency for the controlled bandwidth thread. In the MTNB system, with some sacrifice in latency, the group latency of the controlled bandwidth thread could be reduced. However, this is not possible in the case of the STT system. With buffer depth set at 16, the STT system is less likely to fill up with best-effort threads, preventing priority threads from entering the scheduler. Therefore, the worst case and average case latency of the CPU thread in the STT system is 1.02 and 1.04 times that of MTNB system. However, the deeper queue enables the bursty CPU and best effort threads to more frequently block the controlled bandwidth traffic, so the worst case bandwidth group latency in the STT system is 1.23 times that of the MTNB system.

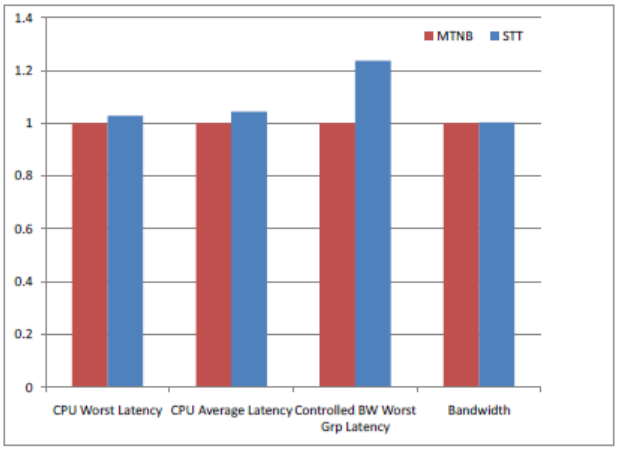
Figure 8 and Figure 9 present similar results for $DTV_{pipelined}$, and Figure 10 and Figure 11 present results for DTV_{stress}. The memory scheduler gives preference to priority threads over jitter sensitive threads. Since there is per-thread flow control, the MTNB system reserves storage to protect the latency

of the CPU thread much more aggressively compared to the STT system. Therefore, we notice that in all our experiments, the worst case latency of the CPU thread is lower for the MTNB case compared to the STT case. On average the STT system had 2.73 times the worst case latency for the CPU thread, 1.66 times the average latency for the CPU thread, and 1.14 times the latency per bandwidth group of the controlled bandwidth traffic, compared to the MTNB system. Moreover, the achieved bandwidth was consistently 1 to 2 % lower than the corresponding MTNB system for all but one case.

8. Conclusion

In this paper, we analyzed two flow control protocols namely, Single Threaded Tag (STT) based protocol, and Multi-Threaded Non-Blocking based protocol. In the paper, we described a typical SoC employing the two protocols and evaluated their relative advantages and disadvantages. We provided arguments supporting the MTNB protocol due to its ability to service priority traffic better than the STT protocol. We evaluated the two protocols by experimentation with representative digital TV (DTV) design and its derivatives. Across all experiments, the MTNB system is able to achieve better performance in terms of servicing the priority and controlled bandwidth traffic, and at the same time, achieves higher operating frequency for the same amount of queuing area.

Figure 6: DTV, Depth=8



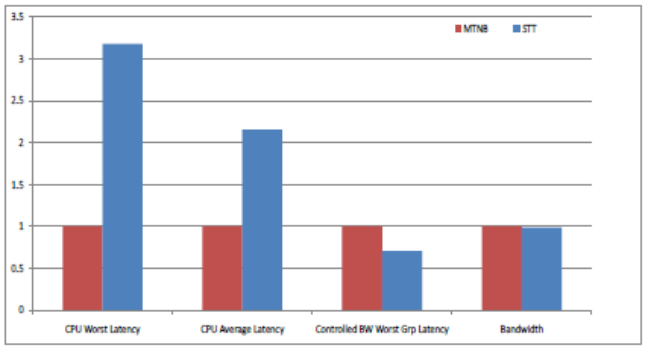


Figure 7: DTV, Depth=16

Figure 8: DTV Pipelined, Depth=16

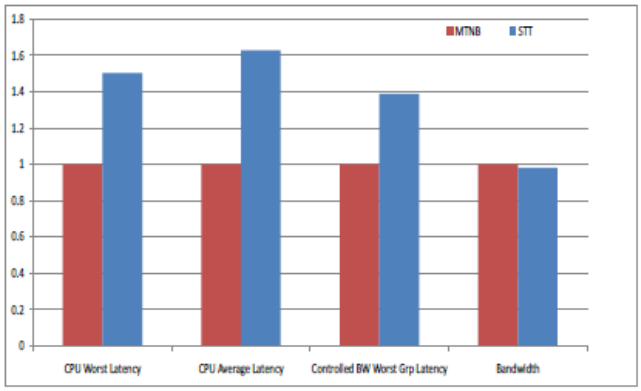


Figure 9: DTV Stress, Depth=8

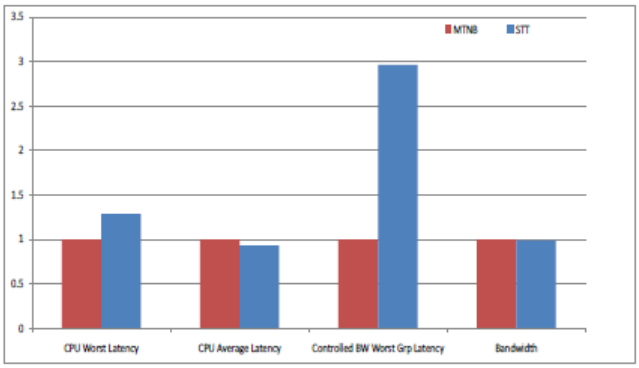
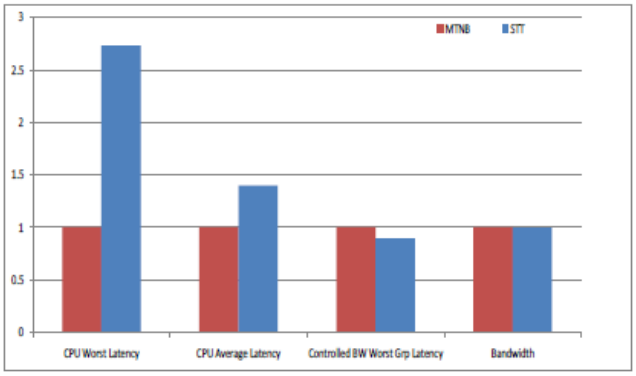


Figure 10: DTV Stress, Depth=16



9. References

[1] *Open Core Protocol Specification, Version 2.2*, available from www.ocpip.org

[2] Chambonniere et al, "MPEG Audio Video Decoder with 2D Graphic Engine for DTV", ICCE 2000

[3] Jose Duato and Sudhakar Yalamanchili, “Interconnection Networks, An Engineering Approach,” Morgan Kaufmann Publishers, 2003

[4] Goossens et al, “Aethereal network-on-chip: concepts, architectures, and implementations ,” IEEE Design and Test of Computers, Volume 22, Issue 5, 2005

[5] Vellanki et al, “Quality-of-Service and Error Control Techniques for Mesh based Network-on-Chip Architectures,” INTEGRATION, the VLSI Journal, Volume 38, 2005

[6] Weber et al, “A quality-of-service mechanism for interconnection networks in system-on-chips,” DATE 2005

[7] Marescaux et al, **“**Introducing the SuperGT network-on-chip: SuperGT QoS: more than just GT,” DAC 2007

[8] Bolotin et al, “QNoC: QoS architecture and design process for Network-on-Chip,” Journal of Systems Architecture, 2003

[9] Srinivasan et al, “Linear Programming based Techniques for Synthesis of Network-on-Chip Architectures,” IEEE Transactions on VLSI, Volume 14, March 2006

[10]AMBA System Architecture, “http://www.arm.com/products/solutions/AMBAHomePage.html”

[11] Sonics, Inc, Milpitas, CA “www.sonicsinc.com”