October 2003

Comparing

Serial ATA Native Command Queuing (NCQ)

and

ATA Tagged Command Queuing (TCQ)

The Advantage of

Serial ATA Native Command Queuing

A WHITEPAPER BY:

Intel Corporation

Summary

Native Command Queuing (NCQ) is among the most anticipated advanced features introduced in the Serial ATA II: Extensions to Serial ATA 1.0 specification, available for download at www.serialata.org. NCQ is designed to increase performance by allowing the hard drive to accept multiple outstanding commands and optimally re-order the execution of those commands.

The general concept of allowing the hard drive the freedom to re-order commands for better performance is not new. SCSI drives have had support for command queuing for more than a decade resulting in superior random I/O performance in queued environments such as in servers and workstations. In 1997, the ATA specification added a Tagged Command Queuing (TCQ) protocol to try and achieve a similar performance benefit on random I/O that SCSI has.

When the Serial ATA 1.0 specification was completed in late 2001, the ATA TCQ protocol was not realizing the full performance potential that command queuing has for queued random I/Os. At that time only one drive manufacturer supported the ATA TCQ protocol and there was virtually no driver support for the feature. One of the reasons for the lack of adoption is that the ATA TCQ protocol has inefficiencies that can cause significant performance degradation in a lightly queued workload common in many desktop environments.

The Serial ATA II Workgroup recognized that improving random I/O performance for Serial ATA drives was important and that the ATA TCQ protocol would not meet this need. The Serial ATA II Workgroup developed the Native Command Queuing (NCQ) protocol to deliver higher random I/O performance while minimizing the inefficiencies that plague the ATA TCQ protocol.

To achieve excellent random I/O performance on highly queued workloads without incurring performance degradations on lightly queued workloads, ensure that the drive and system support *Native* Command Queuing.

Key Feature Comparison

There are three key features that enable a low overhead command queuing protocol – an efficient status return mechanism, low interrupt overhead, and an efficient host memory buffer selection mechanism (referred to as First Party DMA). This section compares the support for and implementation of each of these important features in Native Command Queuing and ATA TCQ.

Status Return Mechanism

A command queuing protocol must define how to tell the host that a command is complete; this is commonly referred to as the status return mechanism. The status return mechanism should allow the drive to efficiently complete commands in an out-of-order fashion.

Native Command Queuing has a status return mechanism that is race-free and without host handshakes. The drive may issue command completions for multiple commands back-to-back or even at the same time. This key feature is possible because the protocol was designed to not require a handshake from the host on a command completion. The status mechanism allows the device to “pile on” completions if two commands finish closely in time and it still allows the host to positively know which of the 32 commands have completed. As an example of the efficiency of this scheme, the drive can return successful status for all 32 commands simultaneously in a single 64-bit packet.

In contrast, the ATA TCQ protocol requires a handshake between the host and device for each and every command completion. To complete a command, the host actually has to issue a new command, called Service, to the device to determine the command to complete. There is no mechanism in ATA TCQ to complete multiple commands at the same time or back-to-back without host intervention. This overhead adds a significant amount of latency to this protocol.

Interrupt Overhead

A key to good performance in any data transfer protocol is minimizing the number of interrupts taken by the host in order to satisfy a request. Each interrupt that is taken increases CPU utilization and latency because the software interrupt service routine does not run immediately. Interrupt service latency can vary from microseconds to milliseconds depending on the current utilization of the system.

Native Command Queuing was designed to minimize the number of interrupts per I/O and has a maximum of one interrupt per command. The number of interrupts per command is often less than one when interrupts are aggregated. Interrupts may be aggregated (combined) by the drive explicitly if it completes multiple commands at the same time. Additionally, if the drive completes multiple commands in a short time span, the individual interrupts for each command are automatically aggregated into one interrupt by the host controller. In a highly queued workload this situation can occur frequently since interrupt service latency may be long in comparison to the time between command completions.

In contrast, ATA TCQ has two interrupts for every command. There is no means to aggregate these interrupts because each step in the ATA TCQ protocol requires a host handshake. There is one interrupt for setting up the DMA engine for a data transfer and another interrupt for completing the command.

Host Memory Buffer Selection (First Party DMA)

Command queuing is designed to allow the drive to select which command to perform a data transfer for next. Allowing the drive to determine the next transfer and directly select the corresponding host memory buffer is called First Party DMA.

Native Command Queuing has First Party DMA support – the drive can directly select the DMA context for a subsequent data transfer without host software intervention. The drive selects the DMA context by sending a packet to the host controller specifying the tag/identifier of the command that the data transfer is for. The host controller hardware then loads the scatter/gather table pointer for that command (based on the tag value) into the DMA engine. Then the DMA transfer may immediately proceed.

ATA TCQ does not support First Party DMA. In the TCQ protocol, the drive causes an interrupt when it is ready to transfer data. The host processes the interrupt and determines that the SERV bit is set in the Status register. This means that the drive is ready to proceed with the data transfer for a particular command. Next the host must issue the Service command to the device to determine the tag of the command the data transfer is for. After the Service command is complete, the host can setup the DMA engine for that command. The interrupt service latency and Service command overhead can be a substantial performance penalty.

Host Controller Interface

To take full advantage of a good queuing protocol, the host controller (programming) interface needs to be geared towards a queued programming model. If the programming interface only allows the driver to issue one command at a time and does not automate DMA engine setup, it will be difficult to achieve maximum benefit from a queuing protocol.

Intel is leading an industry effort to define the Advanced Host Controller Interface (AHCI) for Serial ATA. This programming interface enables software to take advantage of all of the advanced features in Serial ATA 1.0 and Serial ATA II.

AHCI has full support for Native Command Queuing. AHCI provides a “command list” where software can build up to 32 commands with scatter/gather tables for each that the host controller will automatically issue to the device appropriately. This avoids latency and overhead from software being involved in explicitly issuing each and every command. AHCI also includes full

automation for selecting the appropriate DMA engine context when the device issues a First Party DMA request. Thus software is only involved in constructing the command and completing the command to the OS. By keeping software "out of the way" the performance of the solution is increased.

The legacy Bus-master IDE host controller interface is the standard Parallel ATA programming interface and is typically used with ATA TCQ. With this interface, software must explicitly issue each and every command. There is no ability for software to pre-construct the commands to be issued to the device and have the host controller send them automatically. In addition, the interrupt taken to select the DMA engine context in ATA TCQ must be handled explicitly by software. There is a non-deterministic latency until the interrupt service routine can run and then software must issue the Service command to determine the queued command to transfer data for and then must select the appropriate DMA engine context. The non-deterministic overhead associated with programming the DMA engine causes significant performance degradation in a lightly queued workload.

Conclusion

Native Command Queuing is an efficient protocol that has a race-free status return mechanism, low interrupt overhead (<= 1 interrupt per command), and supports First Party DMA. Native Command Queuing addresses the shortcomings of ATA TCQ.

The shortcomings of ATA TCQ include high overhead, two interrupts per command, a large number of host handshakes, and no support for First Party DMA. The issues with ATA TCQ can cause performance degradation in lightly queued workloads where the benefit of queuing is outweighed by the significant overhead of the protocol.

The host controller interface used in a system has a significant impact on the queuing protocol benefits. The Advanced Host Controller Interface for Serial ATA was designed to take full advantage of the Native Command Queuing benefits.

Make sure that your next system has AHCI and supports *Native* Command Queuing to achieve excellent queued random I/O performance.

Author

Amber Huffman, Staff Architect, Intel Corporation

Amber Huffman is a staff architect in a research and development group at Intel where her responsibilities include storage performance and architecture. Amber's current projects concentrate on definition and development of Serial ATA II advanced features and leading the Advanced Host Controller Interface (AHCI) definition. Amber holds a BSE in Computer Engineering from the University of Michigan and has been with Intel for 6 years.

Disclaimer