December 07, 2017

Timeout Mechanism For Latency-Critical Systems

Shu-Yi Yu

TIMEOUT MECHANISM FOR LATENCY-CRITICAL SYSTEMS

ABSTRACT

A system and method are disclosed for managing timeout in latency-critical systems. The method uses a global running timer by which each transaction is given a deadline when it enters the queue, calculated as: deadline = current time + timeout threshold of its QoS (quality of service). The deadline is stored with the corresponding transaction. The total trackable time is divided into 4 quadrants by its 2 most significant bits (MSB). Time wrapping beyond the saturation point is tracked separately from total trackable time. For transactions with deadlines behind the current time but ahead of the saturation point, their deadlines have expired, and these transactions are treated with highest priority. For transactions having deadlines ahead of the current time, their deadlines have not expired, and these transactions can be dispatched according to the bandwidth requirements for access. The current time is incremented at every clock cycle in a continuous progression. This method tracks deadlines of individual pending transactions using less power and cost than prior methods.

BACKGROUND

In many modern system on a chip (SoC) integrated circuits, accessing the dynamic random access memory (DRAM) limits performance of the entire chip. An SoC typically contains multiple agents, such as a central processing unit (CPU), a general processing unit (GPU), an image signal processor (ISP), a display, or the like, all of which need to access DRAM for memory reads and writes. Among these agents, some are latency-critical, and some are bandwidth-intensive. A typical memory controller receives requests from all agents (or clients), and it is crucial to satisfactory performance to honor time requirements while maintaining high bandwidth. Since many factors need to be taken into consideration when arbitrating requests for

DRAM access, a simple first-in-first-out (FIFO) access does not suffice. Various techniques have been used in the past in an attempt to solve time requirements for access. Individual timeout counters are one proposed solution, but these are costly and not always practically feasible. To reduce hardware, a general solution to the timeout problem is to track transaction order instead of actual elapsed time. In this solution, an older transaction is given higher priority. The arbitration is done using the transaction order and other factors, such as agent ID or QoS requirements as arbitration criteria. However, this approach has several disadvantages. First, only the ordering is tracked, so when the entire system is stalled, while the ordering does not change, time elapses and timeout requirements may be exceeded and without being properly acted upon. Second, because most of the ordering registers need to be updated at every scheduling, the solution results in high power consumption.

DESCRIPTION

A system and method are disclosed herein to track the actual elapsed time of each transaction and the corresponding timeout threshold so that latency-critical requests can be scheduled within the time limits. The method uses a global running timer to indicate the current time for various transactions. As shown in FIG. 1, each input transaction is given a deadline when it enters the queue. The calculated deadline is:

deadline = current time + timeout threshold of its QoS (quality of service).

The deadline is stored with the corresponding transaction. The total trackable time is divided into 4 quadrants by its 2 MSBs, as illustrated in FIG. 2. To detect time wrapping, a separate saturation register is used to mark the saturation point. The current time is incremented at every clock cycle.

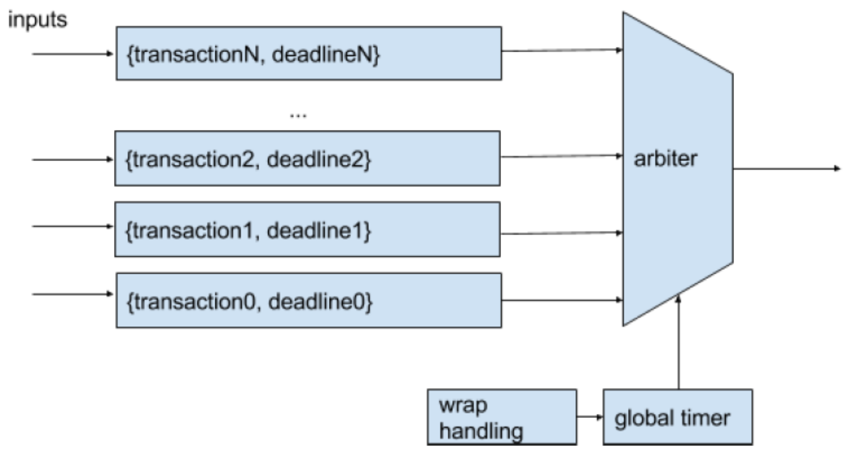
inputs
{transactionN, deadlineN}
...
{transaction2, deadline2}
{transaction1, deadline1}
{transaction0, deadline0}
arbiter
wrap handling
global timer

FIG. 1: Global timer for tracking system-on-chip time

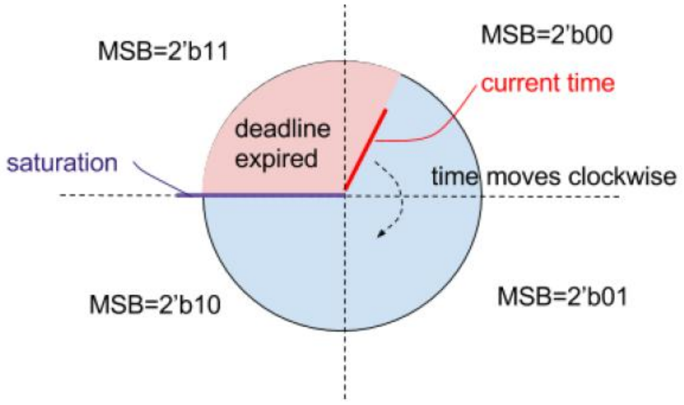

FIG. 2: Division of trackable time into 4 quadrants using a 2-bit MSB

For all the transactions with deadlines ahead of the saturation time but behind the current time, their deadlines have expired and these transactions need to be treated at highest priority. For all the transactions with deadlines ahead of the current time, their deadlines have not expired, and these transactions can be dispatched according to the bandwidth requirements of the system.

At the clock cycle when the current time reaches a new quadrant, the saturation register moves forward one quadrant, and the deadlines falling in the quadrant behind the new saturation limit are marked as saturated, their deadlines reset to the new saturation base. In the specific example of FIG. 2, when the current time reaches quadrant 2'b01, the saturation limit moves to the axis between quadrant 2'b11 and 2'b00, and the transactions remaining in quadrant 2'b11 will be marked as saturated. Any unaccessed transactions in 2’b11 will be reset to be ahead of the new saturation base. In the next clock cycle, tracking continues, new incoming transactions may have their deadlines in the 2’b11 quadrant which will be behind the saturation limit and ahead of the current time, and transactions in the 2b’00 quadrant, which are now behind the current time but ahead of the saturation limit, will have an expired deadline and will be prioritized.

To perform deadline comparison against the current time, all the deadlines are adjusted to the base of saturation. Because time wraps around, for each transaction:

actual deadline = deadline - saturation base; and

actual time = current time - saturation base.

A transaction has timed out if its deadline is smaller than the current time:

timeout = (actual deadline) < (actual time).

For any arbiter that needs to use the actual transaction age or deadline, the age is defined as

age = {saturated, ~actual deadline}.

This allows the arbiter to use a simple comparator and gives the highest priority to the largest age. Depending on the process and frequency, the least significant bits (LSBs) of the timer may be omitted if the arbitration has trouble meeting timing requests.

In an actual example, the maximum programmable timeout threshold can be 511ns. This threshold can be chosen based on a targeted cache hit latency of 256ns. One bit, in addition to the 2 quadrant bits, of the most significant bits can be used to detect the timer wraparound, and the width of the timer can 10 bits. The global timer resets at 0 and counts when there is any pending request. To reduce power consumption, the timer stops when no request is pending. A programmable register PSB_TIMEOUT.SCALE is used to specify how frequently the timer increments. It may count every 1, 2, 4, or 8 clock cycle(s).

The method illustrated presents a novel solution for the classic latency-critical arbitration problem. It uses a single timer with wraparound detection and saturation, to achieve low power and low cost while being able to track deadlines of individual pending transactions. The method may be used in any arbiter that needs to take both bandwidth and latency and other criteria into arbitration consideration.