EECS 700 Project:
TimerQueue Implementation for the HybridThread Framework

Jason Agron
Information and Telecommunication Technology Center - University of Kansas
2335 Irving Hill Road, Lawrence, KS
{jagron}@ittc.ku.edu

Abstract

A TimerQueue core within the HybridThread framework is presented that allows threads within the HybridThread system to make use of the nanosleep() system call. This service provides the capability for threads to put themselves to sleep for a specified amount of time with nanosecond resolution. The overall goal of this project is to design and implement a TimerQueue core that is usable by both SW and HW threads which requires the implementation to take the form of a soft-core IP module resident within the fabric of an FPGA. The core has been designed, implemented, and tested within the HybridThread architecture, and is now ready to be used by system programmers using the HybridThread system. Tests have verified that the TimerQueue core provides the ability for threads to sleep with actual nanosecond resolution along with minimal system overhead and jitter, so as to not affect application execution time.

1 Introduction

The overall goal of this project is to design and implement a soft-core IP module that will provide the capability for threads to sleep with actual nanosecond resolution within the HybridThread framework [3, 6, 5]. The TimerQueue core was designed to reside within the fabric of a FPGA so as to allow the core to constantly monitor an timer-queue without adversely affecting application execution time. If timer-queue management was done this way in software, then there would be little, or possibly no time for application threads to run. Additionally, a TimerQueue within an FPGA allows for a uniform interface to be defined so any sort of computation, whether implemented in HW or SW, can access the services of the TimerQueue core. This core’s services can be accessed via high-level APIs such as the nanosleep() system call, thus abstracting the low-level details of the FPGA implementation in a way familiar to programmers at all levels of expertise.

The rest of this paper presents aspects of the design and implementation of the TimerQueue core, as well as testing results, and evaluations of the performance of the TimerQueue core.

2 Design and Implementation

Initially, it was known that the TimerQueue core must be implemented within a FPGA so that timer-queue management can occur with nanosecond resolution without taking away from application execution time. This means that all of the data structures needed for timer-queue manage-
ment must be resident within the resources of the FPGA in order for the TimerQueue core to operate independently without assistance from the CPU or other execution units. The most important data structure for timer-queue management is the sorted event-queue. This data structure is implemented as a sorted doubly-linked-list using the Block RAMs (BRAMs) within the FPGA. The BRAMs of the FPGA allow a system designer to conserve CLBs for storage at the expense of serializing access to individual storage elements [4]. Linked-list management is already a serial process, so using BRAMs to implement the data structure does not have any negative effects. The BRAMs allow the event-queue to hold entries for every thread in the HybridThread system in a way that provides portability, scalability, and abstraction of the low-level details of data storage within the FPGA.

The doubly-linked-list structure of the event-queue simplifies timer-queue management because the next event to expire is always at the head of the queue. Thus the dequeue operation is very fast and has a constant execution time, however, the enqueue operation does not have a constant execution time because the event-queue must be serially traversed in order to add new entries to the event-queue in sorted order.

Another aspect of the TimerQueue core is the fact that it uses 64-bit values as sleep-deltas. This means that a thread gives a 64-bit delta value for how long it wants to sleep from the current time. Unfortunately, the PowerPC processor as well as the Off-Chip Peripheral Bus (OPB) have data bus widths of only 32-bits. This means that multiple bus transactions are required to "push" the sleep-delta value for a given thread to the TimerQueue core before an entry can be added to the event-queue.

The structure of the event-queue as well as the design constraints resulted in the TimerQueue core having the operations shown in Table 1. These operations are all implemented as atomic memory-mapped operations in the form of read and/or write transactions over the OPB bus.

All incoming operation requests are handled by the TimerQueue core’s slave finite state-machine (SlaveFSM), while event-queue monitoring, thread wakeup and dequeue requests are handled by the TimerQueue core’s master finite state-machine (MasterFSM). A block diagram of the TimerQueue’s structure can be seen in Figure 1.

By implementing the TimerQueue core using VHDL in conjunction with using BRAMs for queue storage, the TimerQueue core is extremely scalable in terms of queue size, as well as portable between different FPGA architectures because it is not directly tied to any strange or unique primitives within the chip. The core is equipped with a generic called C_NUM_THREADS which can alter the number of queue entries stored within BRAM in an easy and efficient manner. This allows the system designer or user to configure the TimerQueue core to their needs in terms of size and thus BRAM usage.

### Table 1. TimerQueue Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>WriteLO(tid, uint32)</td>
<td>Stores the lower 32-bits of the sleep-delta value for a thread</td>
</tr>
<tr>
<td>WriteHI(tid, uint32)</td>
<td>Stores the upper 32-bits of the sleep-delta value for a thread</td>
</tr>
<tr>
<td>Nanosleep(tid)</td>
<td>Inserts the thread’s event entry into the event-queue and puts the thread asleep</td>
</tr>
<tr>
<td>GetDebugReg()</td>
<td>Returns debug information from the TimerQueue core</td>
</tr>
<tr>
<td>Dequeue(tid)</td>
<td>Removes a thread’s event entry from the event-queue as a result of its timer expiring</td>
</tr>
</tbody>
</table>

3 Results

The TimerQueue core has been implemented using VHDL in the framework of the Xilinx [2] EDK toolset. It has been integrated into a Bus-
Functional Model (BFM) Simulation as well as an Xilinx Platform Studio (XPS) Design Project for usage in testing in both simulated and synthesized form.

The BFM simulations allow the entire HybridThread architecture to be simulated from the viewpoint of the CPU. This allows for clock-cycle accurate observations to be made without any intrusion on the HybridThread system. Simulation tests were created that tested the functionality of the internals of the TimerQueue core, as well as its interactions with the rest of the HybridThread system, i.e. wakeup process of threads leaving the event-queue. These tests confirmed the correctness of the TimerQueue core’s operation as well as provided timing results of the basic TimerQueue core functions. The timing results of the hardware operations can be seen in Table 2.

The timing results show that the execution time of all of the TimerQueue operations are extremely deterministic. The only operation whose execution time varies is the Wakeup process operation. This process requires the TimerQueue core to send an add_thread() message to the Thread Manager (TM) which results in adding a thread back to the ready-to-run queue as well as removing the thread’s entry from the event queue via a dequeue() operation. This process can take a variable amount of time because of varying bus arbitration times due to contention for the bus. Simulation results have shown that it takes anywhere from 90 to 100 clock cycles for the entire wakeup process to complete with little to no load on the OPB bus. This variance could be handled if the TimerQueue had a direct interface to the Thread Manager, but this would require that the Thread Manager’s interface to be drastically changed to accomodate this. The delay incurred by the wakeup process can be taken into account when a user is calculating sleep-delta values so that threads using the TimerQueue’s services wakeup at the intended time, with resolution at the granularity of the clock on the FPGA (f = 100 MHz). The TimerQueue core is able to monitor the event-queue at the exact resolution of the FPGA clock which allows for actual timer resolution in the nanosecond range with a near-constant wakeup latency of 100 clock cycles, without affecting application execution time. Event-queue management is done without any need for a periodic system interrupt [7] due to the fact that the
TimerQueue core can run in parallel with threads executing both on the CPU and within the fabric of the FPGA.

Synthesis of the TimerQueue core (including the IPIF bus-attachment) have been performed on a target platform of a Xilinx Virtex-II Pro 30 FPGA attached to an ML310 evaluation board, and the following resource statistics were gathered: 823 out of 13,696 slices, 747 out of 27,392 slice flip-flops, 1,470 out of 27,392 4-input LUTs, 3 out of 136 BRAMs, and a minimum clock period of 8.044 ns (maximum frequency of 124.313 MHz). The TimerQueue core only takes about 2% of the resources of the FPGA and easily meets our on-chip clock frequency of 100 MHz.

The HybridThread system has been synthesized with the TimerQueue core, however, no testing has been done yet due to the fact that 64-bit values are not currently supported with our cross-compiler. This limits the APIs to using 32-bit values and makes them more difficult to use than necessary. Currently, work is being done to enable the usage of 64-bit values with the cross compiler in order to make easy and clear APIs for usage in both user-defined SW and HW threads. The timing results of using the TimerQueue within a synthesized system are expected to be on par with the simulated timing results, although they will last longer due to the fact that the simulated timing results measure raw hardware delay of the core and do not account for delays incurred by caching.

4 Conclusion

The TimerQueue core supplies timed-yield system services in the form of the nanosleep() API. Its policies comply with the POSIX [1] specification for nanosleep(), however the mechanism by which event-queue management is implemented allows for constant monitoring of events in parallel with application execution. This provides actual nanosecond resolution of sleep times in a form that has very low system overhead and jitter. System overhead has been reduced by migrating TimerQueue services into the FPGA which makes them able to run in parallel with the CPU and thus reduces the overhead of traditional queue management. The small amount of jitter is due to the deterministic nature of the FSMs used to implement the TimerQueue core’s operations. The TimerQueue cores services are available for use through high-level APIs that are uniform for threads implemented in either hardware or software.

References