EECS 700 Project Proposal:
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

In this paper, a proposal for building a TimerQueue core within the HybridThread framework will be discussed. The TimerQueue core will allow threads within the HybridThread system to make use of the nanosleep() system call that allows 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.

1 Introduction

Soft-core IP modules, in the world of reconfigurable computing, are equivalent to software libraries in the world of programming and software engineering. These cores provide useful, general purpose functionality that allow for more people to utilize the resources of FPGAs without requiring any experience with hardware design. An entire System-On-A-Chip (SoC) can be built up from a conglomeration of soft-core IP modules and can be programmed using high-level languages (HLLs) such as C++ and Java. The inherent flexibility of a reconfigurable system, as well as the maleability of the modules themselves, allows a system designer or programmer to tailor the reconfigurable fabric of the FPGA to fit their needs exactly [11]. By providing general functionality such as thread management, scheduling, synchronization, and other high-level programming constructs through the use of custom IP cores one can turn a "sea" of un-used gates into a tightly integrated computer system. A system such as this can be used in environments ranging from lightweight embedded sensors on a children's toys all the way to high-end computing clusters used for scientific applications.

In this paper, a TimerQueue core is proposed to be designed and implemented within the architecture set forth by the HybridThread project [5, 8, 7]. The goal is to create a TimerQueue core that implements the nanosleep() system call with true nanosecond resolution for threads resident in both software and hardware. By adding this core to the HybridThread system, threads will now be able to "sleep" for a user-specified amount of time using a familiar high-level API defined by the POSIX standard for nanosleep. The TimerQueue core will need to interact with the existing HybridThread implementations of the Thread Manager and Scheduler which allow for accurate, deterministic, low-overhead, low-jitter thread operations to occur in parallel with other system operations [10, 4]. In order to pro-
vide uniform TimerQueue services to both SW and HW threads, the core must be implemented within the fabric of the FPGA. This also has the positive effect of allowing TimerQueue services to constantly execute in parallel with application processing as well as the processing done by the other cores within the HybridThread system. Once the core has been designed and implemented, all threads in the HybridThread system will be able to use the services provided by the TimerQueue, yet with superior accuracy when compared to that of current software-implementations of the system call. Adding this core to the HybridThread system provides more functionality in terms of high-level APIs that interact with the resources of the FPGA, thus improving the accessibility of reconfigurable computing by providing useful abstractions for those who are unfamiliar with low-level hardware design.

2 Background

Multithreaded systems usually provide support for threads to sleep, or yield, so that other threads can execute in their place. This mechanism allows individual threads to “voluntarily” give up their position on the CPU, or execution unit, and the thread is eventually placed back on the ready-to-run queue so that the thread can be scheduled to run again. The yielding operation can take many forms: it can be timed, it can be untimed, the thread may not yield at all, or a thread may always yield to other threads in the system. The semantics of the different yielding calls are determined by the type of call, i.e. yield() vs. sleep(), as well as the OS-specific implementations of the calls themselves.

For instance, the POSIX specification [2] says the pthread_yield() call has the following semantics:

"The pthread_yield subroutine forces the calling thread to relinquish use of its processor, and to wait in the run queue before it is scheduled again. If the run queue is empty when the pthread_yield subroutine is called, the calling thread is immediately rescheduled.”

While the POSIX specification [1] for the nanosleep() call acts in the following way:

"The nanosleep() function shall cause the current thread to be suspended from execution until either the time interval specified by the argument has elapsed or a signal is delivered to the calling thread, and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time shall not be less than the time specified by the argument, as measured by the system clock.”

One can see that the pthread_yield() call is an untimed yielding operation that immediately puts the calling thread onto the ready-to-run queue, however, if the ready-to-run queue is already empty, the calling thread just continues to run (basically yielding to itself). On the other hand, the nanosleep() call is a timed yielding operation, that allows the thread to be put back on the ready-to-run queue when the thread’s sleep time interval expires or if a signal is delivered to it. This means that a thread that yields using the nanosleep() call can wakeup either before it’s time interval expires, due to system overhead in adding the thread back to the ready-to-run queue, and after it’s expected time interval expires, due to a signal being delivered to the calling thread.

Current operating systems and their associated calls for nanosleep() use timer-queues to manage
asynchronous events. These timer-queues are implemented in software, and thus require CPU time in order to execute. Therefore threaded applications must be interrupted periodically so that timer-queue management can occur. The interrupt rate usually occurs at a rate that is an integer multiple of the system clock and means that the resolution of timer-queue management is roughly the same as that of the system clock. Under Linux, the system clock has a period of 1 ms, which means that timer-queue management will occur, at best, once every millisecond. This means that an expired timer-queue entry may not be handled for 1 ms or more, which introduces a large amount of jitter into a threaded application, as well as blurring the resolution of the nanosleep() call from nanoseconds into the range of milliseconds or worse [9]. If the nanosleep() system call is to be accurately implemented, then the timer-queue must be managed, or monitored, with nanosecond resolution. This is not feasible in software, due to the fact that the overhead of timer-queue management at this rate would not allow for any time for applications to run. All processing time would be solely used to monitor the timer-queue, and no "useful" work would be accomplished. The rest of this paper describes a proposed solution to this problem by designing a TimerQueue core that will be resident within an FPGA so that typical timer-queue management happens in parallel with application execution. This will allow for the TimerQueue core to monitor the timer-queue with nanosecond resolution, while having little to no effect in terms of overhead and jitter on application execution.

3 Proposed Solution/Design

A TimerQueue core for a traditional "all-software" operating system would just serve as an accelerator for timer-queue management operations. However, within the framework of the HybridThread project, a TimerQueue core can interact with the FPGA-based thread manager and scheduler, so that the timer-queue expiration events do not translate into CPU interrupts. Needless interruption of the CPU is prevented by routing all timer-queue expiration events to the Thread Manager and Scheduler. This allows the sleeping thread to be added to the ready-to-run queue in parallel with application execution, and the Scheduler will then decide if the thread running on the CPU should be preempted or not. Furthermore, both SW and HW threads can have uniform access to the TimerQueue core’s services due to the fact that it is resident within the FPGA. This requires that the TimerQueue must be addressable by all threads in the system, so the solution is to wrap a bus-attachment around it, so it can connect to the OPB Core-Connect bus within the FPGA. The bus attachment for the TimerQueue must be both a slave and a master so that it can receive commands from threads, as well as send add_thread commands to the thread manager so that it can place threads onto the ready-to-run queue when their associated time-intervals expire. The bus attachment will provide the interface for accessing the TimerQueue core’s operations. These operations will be accessed by encoding opcodes and parameters in memory-mapped read/write operations that interact with the core. The operations will include a method of setting the sleep-interval as well as a command that causes the thread to go to sleep, or enter the timer-queue.

The queue, within the core itself, must be able to store the time-intervals for all threads in the system, as well as be able to be traversed in a quick-enough manner in order to retain nanosecond resolution for thread sleep times. The queue itself can be implemented in a variety of different ways, but they differ in terms of FPGA resources and space/time considerations. Basically, there are 2 choices in terms of implementation on a Xilinx Virtex-II Pro [3] FPGA: it can be implemented with registers, or it can be implemented using the Block RAMs (BRAMs) within the FPGA. The register-implementation of the
queue will be very fast, because each register can be accessed in parallel, however each register requires CLB resources to be implemented, so the design of the queue itself will take up an enormous portion of the FPGA. The BRAM-implementation of the queue will be quite small and scalable, because storage of interval times does not require any CLB resources, however only 2 entries in BRAM can be accessed at a time which means queue-traversal must be serial, and therefore will take longer. Usage of BRAMs within cores has shown to conserve copious amounts of FPGA resources [6], so a BRAM-implementation of the sorted timer-queue will be built initially, and if simulation results show that this type of queue will not provide adequate performance, then a timer-queue built using registers will be used instead at the expense of taking up much more logic resources within the FPGA. Synthesis of the TimerQueue core, as well as integration into the HybridThread system will be the next step after simulation. In order to perform simulation, synthesis, and integration, the following tools/devices will be needed: ModelSim software, Xilinx’s EDK software, and a Xilinx ML310 evaluation board.

4 Challenges and Issues

The most difficult challenges in the process of designing and implementing the TimerQueue core will be coming up with a queue architecture that allows for actual nanosecond resolution of sleep-intervals, as well as integration of the TimerQueue core into the HybridThread architecture.

The difficulty in queue design comes about because of the differences in which BRAM and registers are accessed. Registers are essentially individual storage cells that have single clock cycle access (often asynchronous reads, synchronous writes), and they can each be read and written in parallel. A register-implementation of a queue would be very fast, but it would take up a lot of space within the FPGA because each of the registers (the queue requires storage for each thread in the system) takes up logical resources within the CLB fabric of the chip. BRAMs are arrays of storage elements, in which 1 or 2 elements can be accessed at a time depending on whether it is a single-port or dual-port BRAM. Also, access to BRAM storage is not as fast as registers: reads take 2 clock cycles, while writes take only 1 clock cycle. Also, with a BRAM-implementation of a queue there is an issue as to whether the queue should be sorted based on time-interval, or should the queue be unsorted, and constantly traversed to find the earliest time-interval that will expire. Keeping a sorted-queue allows for simplified detection of when the next timer-interval expires because only the “head” element of the queue must be looked at, and when the “head” element changes, detection should be re-enabled or re-started. An unsorted-queue requires that a parallel process constantly traverses the queue while looking for the next timer-interval to expire. This means that a timer-interval may expire before it is ”found” which introduces unwanted jitter into the nanosleep() calls functionality. In a sorted-queue, the next timer-interval to expire is always at the “head”, and the next one to expire is “next” in line, so that searching process is minimized to moving the “head” pointer down once in the queue. This allows for next timer-interval to expire to be found as quickly as possible, which minimizes the jitter of the nanosleep() call to that of magnitude of BRAM accesses (1 or 2 clock cycles).

Integration is usually a difficult process in system design, however the HybridThread system uses well-defined memory-mapped interfaces for all of it’s FPGA-resident components, therefore the TimerQueue core can be more easily integrated into the system as long as it abides to the existing interface policy.

With these factors in mind, the initial implementation of the TimerQueue core will contain a sorted timer-queue built using BRAMs. Sim-
ulations will be performed to make sure that this architecture is adequate for providing actual nanosecond resolution for thread sleep times. If this architecture is not adequate then an alternative queue structure will be built using registers to increase core performance at the expense of using more space within the FPGA. Once the core has been instantiated in the system SW and HW APIs will be defined and built so that users can interact with the TimerQueue core at the same high-level of abstraction that normal software-implementations of nanosleep() calls are accessed with. The end result of this project will hopefully provide the HybridThread system with a mechanism for both SW and HW threads to sleep for user-specified times, with actual nanosecond resolution and extremely low levels of jitter.

References